

Stephen Randy Davis



C# pour les Nuls

Publié par
Hungry Minds, Inc.
909 Third Avenue
New york, NY 10022

Pour les Nuls est une marque déposée de Hungry Minds, Inc
For Dummies est une marque déposée de Hungry Minds, Inc
Collection dirigée par Jean-Pierre Cano
Traduction : Philippe Reboul
Édition : Pierre Chauvot
Maquette et illustration : Stéphane Angot

Édition française publiée en accord avec Hungry Minds, Inc.

33, avenue de la République
75011 Paris - France
Tél. 01 40 21 46 46
Fax 01 40 21 46 20
E-mail : firstinfo@efirst.com
Web : www.efirst.com
ISBN : 2-84427-259-2
Dépôt légal : 1er trimestre 2002

Sommaire

Introduction

Au fil des années, les langages de programmation ont beaucoup évolué. Dans les premiers temps, les langages étaient malcommodes et les outils volumineux. Écrire un programme qui fasse quoi que ce soit d'utile était une chose difficile. Au fur et à mesure des progrès de la technologie, des langages plus avancés apparaissaient sur le marché. Il y eut donc, assez rapidement, le langage C, et par la suite C++ (prononcer "C plus plus"). Les outils s'amélioraient aussi. Très vite, il apparut des environnements de développement intégré, avec des éditeurs, des concepteurs, des débogueurs et Dieu sait quoi d'autre, réunis dans des ensembles faits pour vous accompagner du berceau à la tombe.

On pourrait croire que ces nouveaux outils avaient rendu la programmation plus facile, mais il n'en était rien : les problèmes n'en étaient que plus compliqués. C'est juste au moment où je pensais que les programmeurs allaient enfin rattraper ce processus qu'est apparu le développement pour le Web.

Avec l'avènement du Web, le monde s'est divisé en deux camps : les adeptes des solutions basées sur le système d'exploitation Windows, et "les autres". Au début, ce sont "les autres" qui prirent l'avantage. Leurs outils, basés sur le langage Java, permettaient d'écrire des programmes distribués sur le Web.

C'est en juin 2000 que Microsoft a présenté sa réponse, sous la forme d'une famille de langages et d'outils appelée .NET (prononcer "point net", ou "dot net" pour faire américain), avec son emblématique langage de programmation C# (prononcer "C sharp", autrement dit "do dièse"). Bientôt peut-être, on pourra programmer en si bémol majeur !

Les buveurs de Java en revendiquent la supériorité, mais les NETitiens ont aussi leurs arguments. Sans prendre part à leur polémique, on peut dire qu'une bonne partie de la différence peut se résumer en une phrase : Java vous dit qu'il vous suffit de tout réécrire en Java, et vous pourrez exécuter

le résultat sur n'importe quelle machine ; .NET vous dit de ne rien réécrire, et vous pourrez exécuter le résultat sous Windows. (En principe, .NET n'est pas directement lié au système d'exploitation Windows, mais en pratique il y a bien peu de chances que d'autres systèmes d'exploitation importants viennent se placer sous la bannière .NET.)

C# fonctionne au mieux dans l'environnement .NET, permettant de créer des programmes qui communiquent sur le Web, capables notamment de fournir des services à des pages Web existantes. C# peut être intégré à d'autres langages de programmation, comme Visual Basic et Visual C++, permettant aux programmeurs de faire migrer les applications existantes vers le Web sans qu'il soit nécessaire de les réécrire toutes pour cela.

Toutefois, C# n'en est pas moins un langage autonome. Avec l'environnement Microsoft Visual Studio .NET, C# apporte aux programmeurs les instruments dont ils ont besoin pour créer des applications harmonieuses.

Au sujet de ce livre

Ce livre a pour but de vous décrire C#, mais il y a une difficulté.

C# a été créé par Microsoft en tant que partie essentielle de son initiative .NET. Pour des raisons sans doute politiques, Microsoft a soumis au comité de normalisation internationale ECMA au cours de l'été 2000 les spécifications du langage C#, bien avant que .NET ne devienne une réalité. En théorie, n'importe quelle entreprise peut donc proposer sa propre version de C#, écrite pour fonctionner sous n'importe quel système d'exploitation et sur n'importe quelle machine plus grosse qu'une calculatrice.

Toutefois, au moment où j'écris ces lignes, il n'existe qu'un seul fournisseur qui propose un compilateur C# : Microsoft. En outre, Visual C# n'est proposé que d'une seule manière : en tant qu'élément de la suite d'outils Visual Studio .NET.

Aussi, pour vous décrire C#, je ne pourrai éviter de vous parler de Visual Studio, au moins jusqu'à un certain point ; j'ai donc essayé d'en maintenir l'évocation à un minimum raisonnable. Je pourrais me contenter de vous dire : "Ouvrez votre programme de la manière qui vous plaira" ; mais je vous dirai plutôt : "Lancez C# à partir de Visual Studio en appuyant sur la touche F5." Je veux que vous puissiez-vous concentrer sur le langage C# sans avoir à vous casser la tête sur des questions mineures.

D'un autre côté, je suis conscient du fait que beaucoup de lecteurs, sinon la plupart d'entre eux, voudront utiliser C# dans le but d'écrire des applications pour Windows. Bien que ce ne soit pas un livre sur la programmation sous Windows en tant que telle, j'ai consacré une partie à montrer comment C# et Visual Studio forment, ensemble, un puissant environnement de programmation pour Windows.

Je sais aussi que certains utilisateurs se serviront de C# afin de créer des applications distribuées pour le Web ; mais comme on ne peut pas tout mettre dans ce livre, il me faut bien définir une limite quelque part. *C# pour les Nuls* ne s'attaque pas aux questions de .NET et de la programmation distribuée.

Hypothèses gratuites

Avant de pouvoir commencer à programmer en C#, il vous faut avoir installé sur votre ordinateur un environnement de développement C# ; autrement dit, au moment où j'écris, Visual Studio de Microsoft. Pour construire les programmes de ce livre, vous devez avoir installé Visual Studio .NET.

Pour pouvoir seulement exécuter un programme généré avec C#, il faut avoir le Common Language Runtime (CLR). Au cours de sa procédure d'installation, Visual Studio .NET copie le CLR sur votre machine. D'autre part, Microsoft a l'intention d'inclure le CLR dans les versions ultérieures de Windows, mais ne l'a pas encore fait pour le moment.

Comment utiliser ce livre

J'ai fait en sorte que ce livre soit aussi facile à utiliser que possible. Il est déjà bien assez difficile de comprendre un nouveau langage. Inutile de rendre les choses encore plus compliquées. Ce livre est divisé en six parties. Dans la première, je vous présente la programmation en C# avec Visual Studio. Vous y serez guidé étape par étape à travers la création de deux types différents de programme. Je vous encourage fortement à commencer par là en lisant ces deux chapitres avant de vous aventurer dans les autres parties du livre. Même si vous avez déjà écrit des programmes, c'est le schéma de base présenté dans la première partie qui sera utilisé tout au long du livre.

De la deuxième à la quatrième partie, les chapitres sont autonomes. Je les ai écrits de manière que vous puissiez ouvrir le livre au hasard sur n'importe lequel d'entre eux et commencer à lire. Toutefois, si vous êtes un débutant en programmation, il vous faudra commencer par lire la deuxième partie avant de pouvoir passer à la suite. Mais si vous revenez à un sujet particulier pour vous rafraîchir la mémoire, vous ne devriez pas avoir de difficultés à aller directement à la section correspondante sans commencer par lire les 20 pages précédentes.

La cinquième partie revient quelque peu au style "faites comme ceci". *C# pour les Nuls* est un livre sur la programmation en C#, mais c'est en créant de véritables applications pour Windows que C# et Visual Studio .NET brillent de tous leurs feux. Cette partie va donc vous guider à travers les étapes de la construction d'un programme pour Windows, au-delà des choses élémentaires. Une fois que vous aurez tout lu, vous ne saurez pas encore tout sur la construction d'applications Windows puissantes, mais vous aurez appris ce qu'il faut pour partir dans cette direction.

Et bien sûr, la sixième partie termine le livre selon la tradition des livres *Pour les Nuls*.

Comment ce livre est organisé

Voici un bref tour d'horizon de ce que vous allez trouver dans chaque partie :

Première partie : Créer vos premiers programmes C#

Dans votre vie future de programmeur C#, vous allez créer beaucoup de programmes. Quelle meilleure manière de commencer que d'écrire une petite application Windows amusante (j'ai bien dit petite) ? Cette partie va vous montrer, étape par étape, comment écrire la plus petite application Windows possible en utilisant l'interface Visual Studio .NET. Vous apprendrez aussi à créer le cadre de base C# que nous allons utiliser dans le reste du livre.

Deuxième partie : Programmation élémentaire en C#

Dans sa définition la plus élémentaire, une pièce de Shakespeare n'est rien d'autre qu'un ensemble de séries de mots, liées les unes aux autres. D'un point de vue tout aussi élémentaire, 90 % de l'écriture de n'importe quel programme C# consiste en création de variables, en opérations arithmétiques et en instructions de contrôle du chemin d'exécution du programme. Cette partie est consacrée à ces opérations élémentaires.

Troisième partie : Programmation et objets

Déclarer des variables ici et là et faire avec elles des additions et des soustractions est une chose, écrire de véritables programmes pour de véritables utilisateurs en est une autre. La troisième partie est consacrée à la manière d'organiser vos données pour les rendre plus faciles à utiliser dans la création d'un programme.

Quatrième partie : La programmation orientée objet

Vous pouvez toujours organiser les différentes parties d'un avion comme vous voulez, mais tant que vous ne serez pas arrivé à lui faire faire quelque chose, ce ne sera rien d'autre qu'une collection de parties. Il pourra aller quelque part seulement lorsque vous l'aurez fait décoller.

C'est sur la base du même principe que la quatrième partie va vous expliquer comment transformer une collection de données en un véritable objet. Un objet qui contient différents éléments, bien sûr, mais qui peut imiter les propriétés d'un objet du monde réel. Cette partie présente donc l'essence de la programmation orientée objet.

Cinquième partie : Programmer pour Windows avec Visual Studio

Il ne suffit pas d'avoir compris le langage C# pour savoir écrire une application Windows complète avec toutes sortes de fonctions, de boutons et autres raffinements. Rien que pour le plaisir, la cinquième partie

vous guide dans l'utilisation de C# avec l'interface Visual Studio afin de créer une application Windows "non élémentaire". Vous serez fier du résultat, même si vos enfants n'appellent pas leurs copains pour le voir.

Sixième partie : Petits suppléments par paquets de dix

C# est très doué pour trouver des erreurs dans vos programmes. Par moment, je le trouve même un peu trop empressé à me faire remarquer mes faiblesses. Mais, croyez-le ou non, il fait ça pour vous rendre service. Il vous fait remarquer des problèmes que vous auriez dû découvrir vous-même s'il n'avait pas été là pour ça.

Malheureusement, les messages d'erreur peuvent être un peu confus. L'un des chapitres de cette partie présente les messages d'erreur de génération C# les plus courants, leur signification, et la manière de s'en débarrasser.

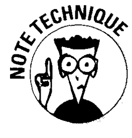
De nombreux lecteurs viendront à C# avec l'expérience antérieure d'un autre langage de programmation. Le deuxième chapitre de cette partie expose les dix principales différences entre C# et son géniteur, C++.

Au sujet du site Web

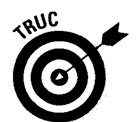
Sur notre site Web, vous trouverez tout le code source contenu dans ce livre. Rendez-vous sur le site des éditions First à l'adresse `www.efirst.com`. Une fois sur la page d'accueil, cliquez sur First Interactive, puis sur la rubrique Téléchargement. Ensuite, faites défiler les ouvrages jusqu'à *C# Pour les Nuls*, cliquez sur le lien pour télécharger le fichier ZIP contenant l'ensemble des fichiers, et décompressez-le dans un répertoire de votre choix.

Icônes utilisées dans ce livre

Tout au long de ce livre, j'utilise les icônes suivantes pour mettre en évidence des informations importantes.



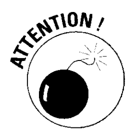
Cette icône indique des aspects techniques que vous pouvez ignorer en première lecture.



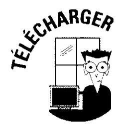
L'icône Truc signale une information qui peut vous épargner pas mal de temps et d'efforts.



Souvenez-vous de cela. C'est important.



Souvenez-vous aussi de ce qui est indiqué par cette icône. C'est le genre de chose qui vous tombe dessus au moment où vous vous y attendez le moins et qui peut produire un bogue vraiment difficile à débusquer.



Cette icône identifie le code que vous trouverez sur le site des éditions First. Vous y gagnerez quelques efforts de frappe au clavier, mais n'en abusez pas. Vous comprendrez mieux C# en saisissant les programmes vous-même.

Conventions utilisées dans ce livre

Pour faciliter les choses, j'ai utilisé différentes conventions. Les termes qui ne sont pas des "mots ordinaires" apparaissent dans `cette police`, afin de réduire au minimum les risques de confusion. Les listings de programmes sont mis en retrait dans le texte de la façon suivante :

```
use System;
namespace MyNameSpace
{
  public class MyClass
  {
  }
}
```

Chaque listing est suivi par une explication subtile et profonde. Les programmes complets sont en téléchargement sur le site des éditions First, ce qui fera votre bonheur, mais les petits fragments de code n'y sont pas.

Enfin, vous verrez des séquences d'ouverture de menus comme dans "Sélectionnez Fichier/Ouvrir avec/Bloc-notes", ce qui signifie : cliquer sur le menu Fichier, puis, dans le menu qui apparaît, sur Ouvrir avec, et enfin, dans le sous-menu qui apparaît, de sélectionner Bloc-notes.

Où aller maintenant

Naturellement, la première étape est de comprendre le langage C#, idéalement en lisant *C# pour les Nuls*. En ce qui me concerne, je m'accorderais quelques mois pour écrire des programmes C# simples avant de passer à l'étape suivante qui est d'apprendre à créer des applications Windows. La cinquième partie pourrait faire paraître les choses faciles, mais il y a pas mal de pièges. Essayez tous les composants disponibles dans la boîte à outils de Visual Studio. Son système d'aide en ligne, très complet et pratique, les décrits tous. Accordez-vous un bon nombre de mois d'expérience de création d'applications Windows avant de vous lancer dans l'écriture de programmes destinés à être distribués sur Internet.

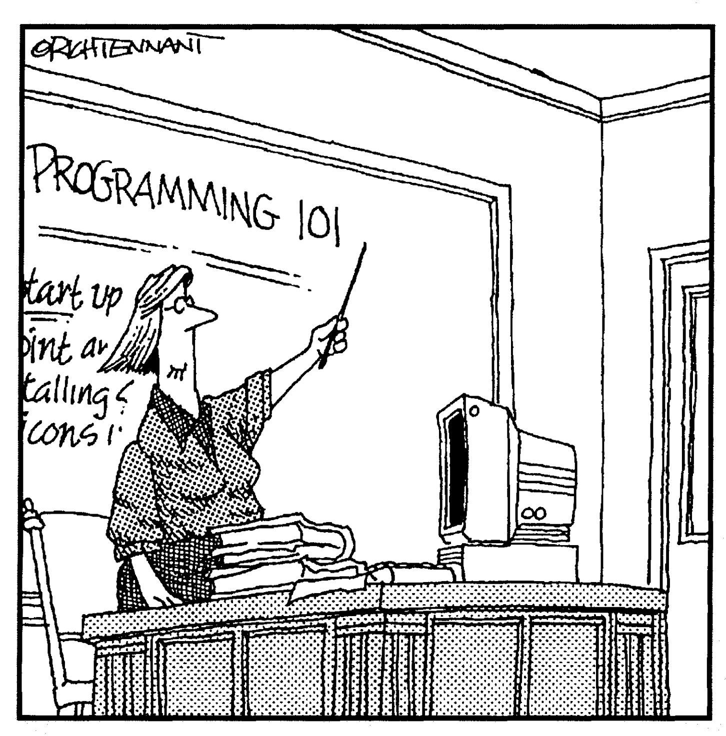
Entre-temps, vous disposez de plusieurs endroits pour vous tenir au courant de l'actualité de C#. Pour commencer, tournez-vous vers la source officielle : `msdn.microsoft.com`. Il existe aussi de nombreux sites Web de programmeurs qui contiennent des éléments très complets sur C#, et qui permettent aussi de participer à des discussions en ligne sur les sujets les plus divers, de la manière d'enregistrer un fichier source aux mérites combinés des ramasse-miettes (garbage collectors) déterministes et non déterministes. Voici quelques grands sites sur C#, sans ordre particulier :

- `www.codeguru.earthweb.com/csharp`
- `csharpindex.com`
- `www.c-sharpcorner.com`

J'ai aussi mon propre site Web, `www.stephendavis.com`, qui contient une liste de questions fréquemment posées (FAQ, Frequently Asked Questions). S'il y a une chose que vous n'arrivez pas à comprendre, la réponse à ce qui vous préoccupe s'y trouve peut-être déjà. J'y ai aussi ajouté une liste de toutes les erreurs qui ont pu se glisser dans le livre. Enfin, il y a un lien vers mon adresse de messagerie qui vous permettra de m'envoyer un mail si vous ne trouvez pas ce que vous cherchez.

Première partie

Créer vos premiers programmes C#



"Avant d'aborder les aspects avancés comme la fonction 'EjecterLesTouristesQuiNeSuiventPas', nous allons commencer par les principes de base."

Dans cette partie...

D'ici à ce que vous ayez maîtrisé C#, vous avez pas mal de chemin à faire. Autant commencer par vous amuser un peu. Cette première partie va vous montrer les étapes de la création d'une application Windows aussi simple que possible en utilisant l'interface de Visual Studio .NET. Vous y apprendrez aussi à créer le cadre de travail de base en C# pour les exemples de programmes qui apparaissent tout au long de ce livre.

Chapitre 1

Créer votre premier programme C# pour Windows

Dans ce chapitre :

Qu'est-ce qu'un programme ? Qu'est-ce que C# ? Où suis-je ?

Créer un programme pour Windows.

Bien accorder votre environnement Visual Studio .NET pour C#.

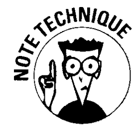
Dans ce chapitre, je vais donner quelques explications sur les ordinateurs, les langages de programmation, C#, et Visual Studio .NET. Ensuite, je vous guiderai à travers les étapes de la création d'un programme pour Windows très simple, écrit en C#.

Les langages de programmation, C#, et .NET

Un ordinateur est un serviteur remarquablement rapide, mais remarquablement stupide. Il fera tout ce que vous lui demanderez (dans la limite de ses capacités) très vite, et même de plus en plus vite. À l'heure actuelle, un microprocesseur d'usage courant pour PC est capable de traiter près d'un milliard d'opérations par seconde.

Malheureusement, un ordinateur ne comprend rien de ce qui ressemble à un langage humain. Vous pouvez toujours me dire : "Mon téléphone compose le numéro de la personne que je veux appeler si je lui dis son nom. Je sais qu'il y a un petit ordinateur qui pilote mon téléphone. Donc, cet ordinateur parle français." En fait, c'est un programme qui interprète ce que vous dites, pas l'ordinateur lui-même.

Le langage de l'ordinateur est souvent appelé *langage machine*. Pour un être humain, il est possible, mais extrêmement difficile et fertile en erreurs, d'écrire en langage machine.



Pour des raisons historiques, le langage machine est aussi appelé langage d'assemblage. Chaque constructeur fournissait avec ses machines un programme nommé assembleur qui convertissait des mots particuliers en instructions du langage machine. Ainsi, vous pouviez écrire des choses vraiment cryptiques du genre `MOV AX,CX` (c'est une véritable instruction pour processeur Intel), et l'assembleur convertissait cette instruction en une suite de bits correspondant à une seule instruction machine.

Les êtres humains et les ordinateurs ont décidé de se rencontrer quelque part entre les deux. Les programmeurs écrivent leurs programmes dans un langage qui est loin d'être aussi libre que le langage humain, mais beaucoup plus souple et plus facile à utiliser que le langage machine. Les langages qui occupent cette zone intermédiaire (par exemple C#) sont appelés langages de *haut niveau* (le terme *haut* a ici un sens relatif).

Qu'est-ce qu'un programme ?

Qu'est-ce qu'un programme ? Avant tout, un programme pour Windows est un fichier exécutable que l'on peut lancer en double-cliquant sur son icône dans une fenêtre. Par exemple, la version du traitement de texte Word que j'utilise pour écrire ce livre est un programme. On peut appeler cela un *programme exécutable*, ou tout simplement un *exécutable*. Le nom du fichier d'un programme exécutable se termine généralement par l'extension .EXE.

Mais un programme est aussi autre chose. Un programme exécutable comporte un ou plusieurs *fichiers source*. Un fichier de programme C# est un fichier texte qui contient une séquence de commandes C#, se suivant selon les règles de la syntaxe de C#. On appelle fichier source un tel fichier, probablement parce que c'est une source de frustration et d'angoisse.

Qu'est-ce que C# ?

Le langage de programmation C# est l'un de ces langages intermédiaires qu'utilisent les programmeurs pour créer des programmes exécutables. C# comble le fossé qui existait entre le puissant mais compliqué C++ et le facile mais limité Visual Basic. Un fichier de programme C# porte l'extension .CS.

C# est :

- **Souple :** Un programme C# peut être exécuté sur la machine sur laquelle il se trouve ou bien transmis par l'intermédiaire du Web pour être exécuté sur un ordinateur distant.
- **Puissant :** C# dispose essentiellement du même jeu d'instructions que C++, mais avec les angles arrondis.
- **Facile à utiliser :** Dans C#, les commandes responsables de la plupart des erreurs dans C++ ont été modifiées pour les rendre plus sûres.
- **Visuel :** La bibliothèque de C# fournit les outils nécessaires pour créer directement des fenêtres d'affichage élaborées, avec des menus déroulants, des fenêtres à onglets, des barres de défilement et des images d'arrière-plan, entre autres.
- **Prêt pour Internet :** C# est le pivot de la nouvelle stratégie Internet de Microsoft, nommée .NET (prononcer *point net*).
- **Sûr :** Tout langage destiné à une utilisation sur Internet doit contenir sous une forme ou sous une autre des outils de sécurité pour se protéger contre les hackers.

Enfin, C# est une partie intégrante de .NET.

Qu'est-ce que .NET ?

.NET est la stratégie adoptée par Microsoft dans le but d'ouvrir le Web aux simples mortels comme vous et moi. Pour comprendre cela, il vous faut en savoir un peu plus.

Il est très difficile de programmer pour Internet dans des langages un peu anciens comme C ou C++. Sun Microsystems a répondu à ce problème en créant le langage Java. Celui-ci repose sur la syntaxe de C++, rendue un peu plus accessible, et est centré sur le principe d'un développement distribué.

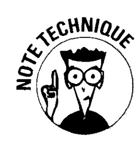


Quand un programmeur dit "distribué", il pense à des ordinateurs dispersés géographiquement, exécutant des programmes qui se parlent les uns aux autres, dans la plupart des cas par Internet.

Microsoft a décidé de se lancer dans la course et a acquis une licence du code source de Java, créant sa propre version nommée Visual J++ (prononcer "J plus plus"). Microsoft obtint ainsi un accès instantané aux progrès accomplis par Sun et de nombreuses autres entreprises en développant des utilitaires en Java. Il y eut toutefois quelques problèmes lorsque Microsoft tenta d'ajouter des fonctions à Java, car son contrat de licence du code source le lui interdisait. Pire encore, le contrat était si simple qu'il était impossible d'y lire autre chose que ce qu'on avait voulu y mettre. Sun avait réussi à bouter Microsoft hors du marché Java.

Il était finalement aussi bien de se retirer de Java, parce qu'il avait un sérieux problème : pour en tirer tous les avantages, il y avait intérêt à écrire tout son programme en Java. Comme Microsoft avait trop de développeurs et trop de millions de lignes de code source existantes, il lui fallait inventer un moyen de prendre en compte plusieurs langages. C'est ainsi que .NET vint au monde.

.NET est un cadre de travail, en bien des points semblable à celui de Java.



La plate-forme de la génération précédente était constituée d'outils aux noms étranges, comme Visual C++ 6.0, COM+, ASP+, Dynamic Linked Libraries et Windows 2000 (et versions antérieures). .NET leur apporte Visual Studio .NET, une amélioration de COM+, ASP.NET, une nouvelle version de Windows, et des serveurs prenant en compte .NET. .NET quant à lui prend en compte les nouveaux standards de communication comme XML et SOAP, plutôt que les formats propriétaires de Microsoft. Enfin, .NET prend en compte le dernier grand mot d'ordre qui fait fureur, comme en son temps l'orientation objet : les services Web.

Microsoft revendique volontiers que .NET est très supérieur à la suite d'outils pour le Web de Sun, basée sur Java, mais la question n'est pas là. Contrairement à Java, .NET ne vous demande pas de réécrire vos programmes existants. Un programmeur Visual Basic peut se contenter d'ajouter à son programme quelques lignes de C# afin de le rendre "bon pour le Web" (ce qui signifie qu'il sait se procurer des données sur Internet). .NET prend en compte tous les langages de Microsoft, plus une vingtaine de langages d'autres origines, mais c'est bien C# qui est le navire amiral de la flotte .NET. Contrairement à la plupart des autres langages, C# peut accéder à toutes les fonctions de .NET.

Qu'est-ce que Visual Studio .NET ? et C# ?

Vous vous posez sûrement beaucoup de questions. Le premier langage de programmation populaire de Microsoft a été Visual C++, ainsi nommé parce qu'il avait une interface utilisateur graphique (ou GUI, Graphical User Interface). Celle-ci contenait tout ce dont on pouvait avoir besoin pour développer des programmes C++ bien ficelés.

Puis Microsoft a créé d'autres langages de type "Visual", notamment Visual Basic et Visual FoxPro, pour finalement les intégrer tous dans un même environnement : Visual Studio. Visual Studio 6.0 se faisant de moins en moins jeune, les développeurs en attendaient avec impatience la version 7. C'est peu après le lancement de celle-ci que Microsoft a décidé de la renommer Visual Studio .NET, de manière à mettre en évidence la relation entre ce nouvel environnement et .NET.

D'abord, j'ai plutôt pris ça pour un stratagème, jusqu'au moment où j'ai commencé à l'examiner sérieusement. Visual Studio .NET est assez significativement différent de ses prédécesseurs, suffisamment pour justifier un nouveau nom.



Microsoft a nommé Visual C# son implémentation du langage C#. En réalité, ce n'est rien d'autre que le composant C# de Visual Studio. C# est C#, avec ou sans Visual Studio.

Et voilà. Plus de questions.

Créer une application pour Windows avec C#

Pour vous aider à vous mettre dans le bain avec C# et Visual Studio, cette section va vous conduire à travers les étapes de la création d'un programme Windows. Un programme Windows est couramment appelé une application Windows, plus familièrement WinApp. Notre première WinApp nous servira de schéma de base pour les programmes Windows que nous allons créer par la suite.

En outre, ce programme va vous servir de test pour votre environnement Visual Studio. Ce n'est qu'un test, mais c'est aussi un véritable programme Windows. Si vous réussissez à créer, générer et exécuter ce programme, alors votre environnement Visual Studio est correctement configuré, et vous êtes prêt à lever l'ancre.

Créer le modèle

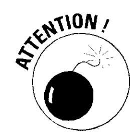
Écrire une application Windows à partir de zéro est un processus difficile, c'est bien connu. Il y a beaucoup de gestionnaires de sessions, de descripteurs, de contextes, beaucoup de défis à relever, même pour un programme simple.

Visual Studio .NET en général et C# en particulier simplifient considérablement la tâche de création d'une application Windows, même déjà très simple. Pour être franc, je regrette un peu que vous ne soyez pas obligé de tout faire à la main. Si le cœur vous en dit, vous pouvez essayer avec Visual C++... Mais je n'insiste pas.

Comme le langage C# est conçu spécialement pour faire des programmes qui s'exécutent sous Windows, il peut vous épargner bien des complications. En plus, Visual Studio .NET comporte un Assistant Applications qui permet de créer des modèles de programme.

Typiquement, un *modèle de programme* ne fait rien par lui-même, en tout cas rien d'utile (un peu comme la plupart de mes programmes), mais il vous fait passer sans effort le premier obstacle du démarrage. Certains modèles de programme sont raisonnablement sophistiqués. En fait, vous serez bien étonné de tout ce que l'Assistant Applications est capable de faire.

Pour commencer, lancez Visual Studio .NET.



N'oubliez pas qu'il faut d'abord avoir installé Visual Studio.

1. **Pour lancer Visual Studio, cliquez sur Démarrer/Programmes/ Microsoft Visual Studio.NET 7.0/Microsoft Visual Studio.NET 7.0, comme le montre la Figure 1.1.**

   Le CPU s'agite, le disque de même, et Visual Studio apparaît. C'est ici que les choses deviennent intéressantes.

2. **Sélectionnez Fichier/Nouveau/Projet, comme le montre la Figure 1.2.**

   Visual Studio ouvre la boîte de dialogue Nouveau projet, comme le montre la Figure 1.3.



   Un *projet* est une collection de fichiers que Visual Studio assemble pour en faire un seul programme. Tous vos fichiers seront des fichiers source C#, portant l'extension .CS. Un fichier de projet porte l'extension .PRJ.

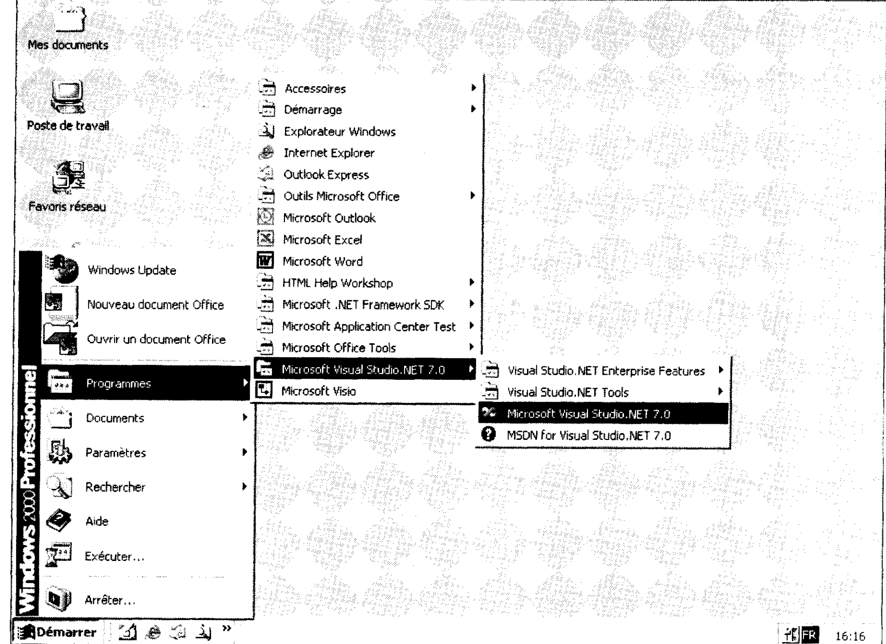


Figure 1.1 : La ligne droite n'est pas le plus court chemin du Bureau à C#.

3. **Dans le volet Types de projets, sélectionnez Projets Visual C#, et dans le volet Modèles, sélectionnez Application Windows. Si vous ne voyez pas la bonne icône de modèle, ne vous inquiétez pas. Faites défiler le contenu du volet Modèles pour la faire apparaître.**

   Ne cliquez pas encore sur OK.

4. **Dans le champ Nom, entrez un nom pour votre projet ou laissez le nom par défaut.**

   L'Assistant Applications va créer un dossier dans lequel il va stocker différents fichiers, notamment le fichier source initial C# du projet. L'Assistant Applications utilise comme nom de ce dossier le nom que vous avez entré dans le champ Nom. Le nom initial par défaut est `WindowsApplication1`. Si vous l'avez déjà utilisé pour un projet, il devient `WindowsApplication2` ou `WindowsApplication3`, et ainsi de suite.

   Pour cet exemple, vous pouvez utiliser le nom par défaut ainsi que l'emplacement par défaut pour le nouveau dossier : Mes documents\Projets Visual Studio\`WindowsApplication1`.

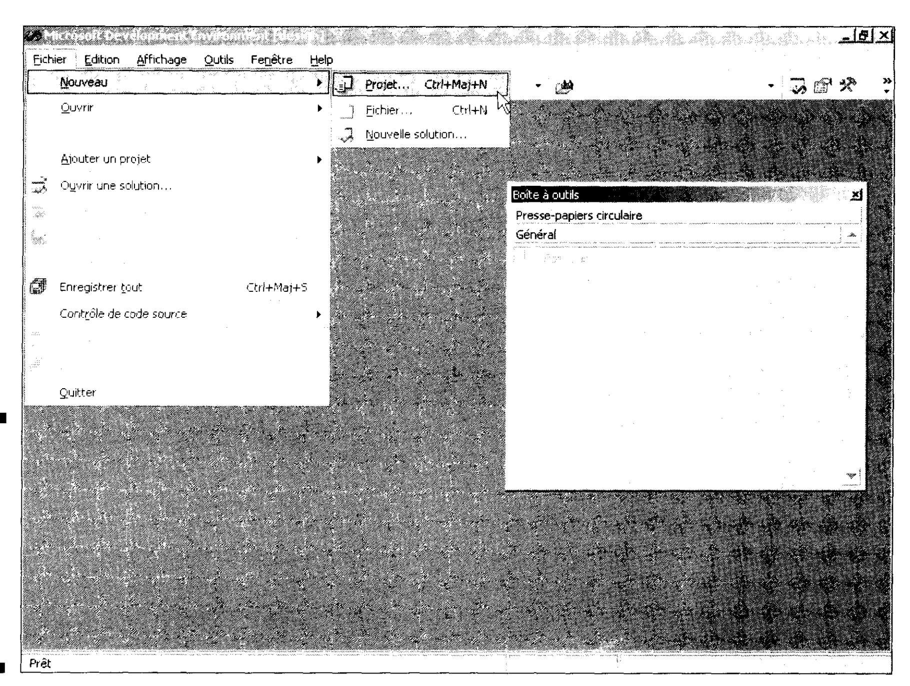


Figure 1.2 : Créer un nouveau projet vous met sur la voie d'une application Windows.

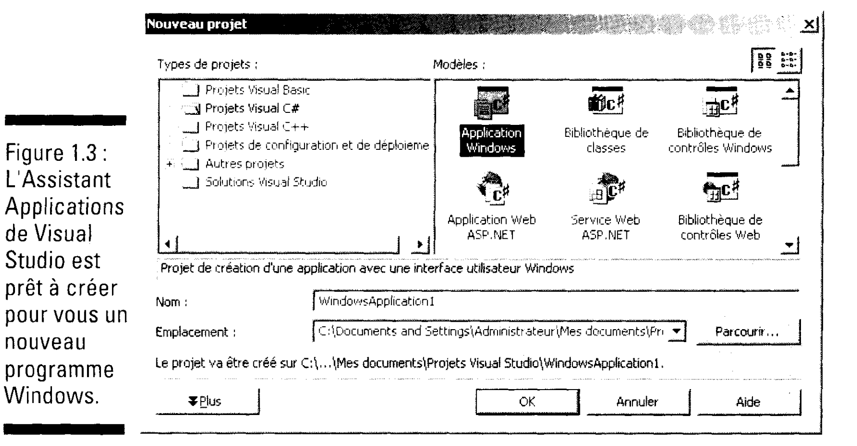


Figure 1.3 : L'Assistant Applications de Visual Studio est prêt à créer pour vous un nouveau programme Windows.

5. **Cliquez sur OK.**

   Le disque dur s'agite quelques instants (parfois longs) avant d'ouvrir un cadre vierge nommé *Form1* au milieu de la fenêtre Visual Studio.

Générer et exécuter votre premier véritable programme Windows

Une fois que l'Assistant Applications a chargé le modèle de programme, Visual Studio ouvre le programme en mode Conception. Vous devez convertir en application Windows ce programme source C# vide, rien que pour vous assurer que l'application initiale créée par l'Assistant Applications ne contient pas d'erreurs.



On appelle *générer* (build) l'acte de convertir un fichier source C# en une véritable application Windows en état de fonctionner. Si votre fichier source contient des erreurs, Visual C# les trouvera dans le processus de construction.

Sélectionnez Générer/Générer. Une fenêtre de résultats s'ouvre, dans laquelle défile une succession de messages. Le dernier de ces messages doit être `Génération : 1 a réussi, 0 a échoué, 0 a été ignoré.`

La Figure 1.4 montre à quoi ressemble mon Bureau après la génération de l'application Windows par défaut. Vous pouvez déplacer les fenêtres comme vous voulez. Les plus importantes sont la fenêtre Form1.cs[Design] et la fenêtre Sortie.

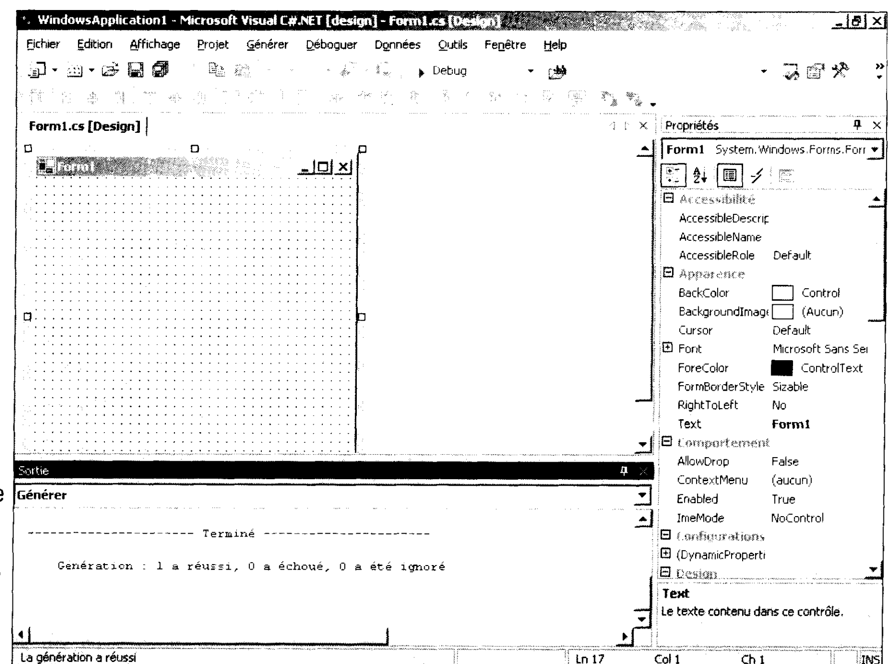


Figure 1.4 : Le modèle de programme initial pour Windows n'a rien de bien excitant.

Vous pouvez maintenant exécuter ce programme en sélectionnant Déboguer/ Exécuter sans débogage. Lorsque vous le lancez, ce programme doit ouvrir une fenêtre exactement semblable à la fenêtre Form1.cs[Design], mais sans les points, comme le montre la Figure 1.5.

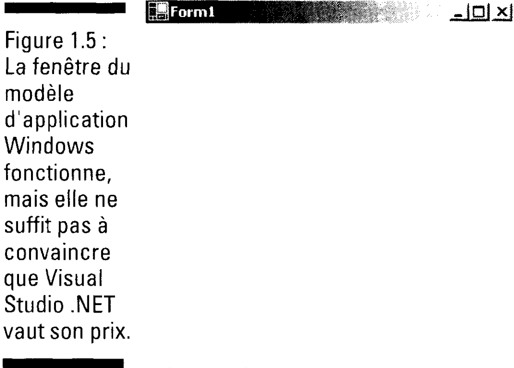
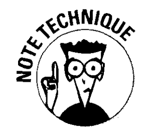


Figure 1.5 : La fenêtre du modèle d'application Windows fonctionne, mais elle ne suffit pas à convaincre que Visual Studio .NET vaut son prix.

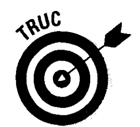


Dans la terminologie C#, cette fenêtre s'appelle un *formulaire*. Un formulaire est doté d'un cadre, avec en haut une barre de titre contenant les boutons Réduire, Agrandir, et Fermer.

Pour arrêter le programme, cliquez sur le bouton Fermer dans le coin supérieur droit de la fenêtre.

Vous voyez : il n'est pas si difficile de programmer en C#.

Indépendamment de ce qu'il fait, ce programme initial est un test pour votre installation. Si vous êtes parvenu jusqu'ici, alors votre environnement Visual Studio est dans l'état qui convient aux programmes que nous allons voir dans la suite de ce livre.



Pendant que vous y êtes, mettez donc à jour votre CV pour y faire savoir que vous êtes officiellement un programmeur d'application Windows. Pour le moment, vous pouvez vous contenter de mettre "application" au singulier.

Dessiner une application

Le programme Windows par défaut n'est pas bien excitant, mais vous pouvez l'améliorer un peu. Revenez dans Visual Studio, et sélectionnez l'onglet Form1.cs[Design]. C'est la fenêtre du Concepteur de formulaires.

Le Concepteur de formulaires est un outil très puissant. Il vous permet de "dessiner" vos programmes dans le formulaire. Une fois que vous avez terminé, cliquez sur Générer, et le Concepteur de formulaires crée le code C# nécessaire pour réaliser une application avec le joli cadre que vous venez de dessiner.

Dans les sections suivantes, vous allez générer une application avec deux champs de texte et un bouton. L'utilisateur peut saisir ce qu'il veut dans l'un des champs de texte (la source), mais pas dans l'autre (la cible). Lorsque l'utilisateur clique sur un bouton intitulé Copier, le programme copie le texte du champ source dans le champ cible.

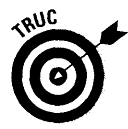
Mettre en place quelques contrôles

L'interface utilisateur de Visual Studio est constituée de différentes fenêtres. Tous les éléments comme les boutons et les zones de texte sont des *contrôles*. Afin de créer un programme Windows, vous allez utiliser ces outils pour en réaliser l'interface utilisateur graphique (GUI), qui est généralement la partie la plus difficile à réaliser d'un programme Windows. Dans le Concepteur de formulaires, ces outils se trouvent dans une fenêtre nommée Boîte à outils.

Si la Boîte à outils n'est pas ouverte, sélectionnez Affichage/Boîte à outils. La Figure 1.6 montre cette Boîte à outils.



Figure 1.6 : La Boîte à outils de Visual Studio contient une quantité de contrôles utiles.



Si vos fenêtres ne sont pas aux mêmes endroits que les miennes, ne vous inquiétez pas. Votre Boîte à outils peut très bien se trouver à gauche, à droite ou au milieu de l'écran. Vous pouvez déplacer chaque fenêtre où vous voulez dans la fenêtre Visual Studio.

La Boîte à outils comporte plusieurs sections, dont Données, Composants, et Windows Forms (qui sera peut-être devenue "Formulaires Windows" dans la version que vous aurez entre les mains). Ces sections permettent simplement d'organiser les contrôles afin que vous puissiez les trouver plus facilement. La Boîte à outils contient de très nombreux contrôles, et vous pouvez aussi créer les vôtres.

Dans la Boîte à outils, cliquez sur `Windows Forms`. Ces contrôles vont vous permettre d'améliorer vos formulaires. Vous pouvez utiliser les petites flèches que vous voyez à droite pour faire défiler la liste.

Pour ajouter un contrôle dans un formulaire, il suffit de le faire glisser et de le déposer à l'endroit voulu. Essayez :

1. **Faites glisser le contrôle Textbox sur le formulaire `Form1`, et relâchez le bouton de la souris.**

   Une zone de texte apparaît dans le formulaire, contenant le texte `textBox1`. C'est le nom assigné à ce contrôle par le Concepteur de formulaires. Vous pouvez redimensionner la zone de texte en cliquant dessus et en faisant glisser ses poignées.



   Vous ne pouvez augmenter que la longueur d'une zone de texte, pas sa hauteur, car par défaut une zone de texte ne comporte qu'une seule ligne.

2. **Faites glisser une autre zone de texte et déposez-la au-dessous de la première.**

3. **Faites maintenant glisser un bouton et déposez-le au-dessous des deux zones de texte.**

4. **Redimensionnez le formulaire et déplacez les contrôles que vous venez d'y mettre jusqu'à ce que le résultat vous convienne.**

   La Figure 1.7 montre mon formulaire.

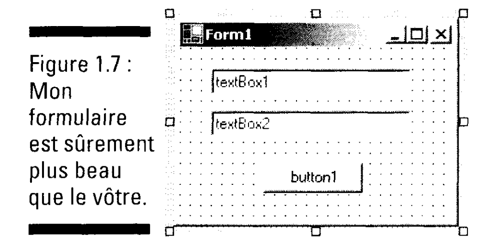


Figure 1.7 : Mon formulaire est sûrement plus beau que le vôtre.

Maîtriser les propriétés

Le problème le plus flagrant de cette application est maintenant que l'étiquette du bouton qu'elle contient, `button1`, n'est pas très descriptive. Nous allons commencer par y remédier.

Chaque contrôle possède un ensemble de propriétés qui en déterminent l'apparence et le fonctionnement. Vous pouvez y accéder par la fenêtre Propriétés :

1. **Sélectionnez le bouton en cliquant dessus.**

2. **Faites apparaître la fenêtre Propriétés en sélectionnant Affichage/ Fenêtre Propriétés.**

   Le contrôle Button possède plusieurs ensembles de propriétés, dont les propriétés d'apparence, qui apparaissent dans la partie supérieure de la fenêtre Propriétés, et les propriétés de comportement, qui apparaissent au-dessous. C'est la propriété Text que vous devez changer.

3. **Dans la colonne de gauche de la fenêtre Propriétés, sélectionnez la propriété Text. Dans la colonne de droite, tapez** Copier, **et appuyez sur Entrée.**

   La Figure 1.8 montre ces paramètres dans la fenêtre Propriétés, et le formulaire qui en résulte.

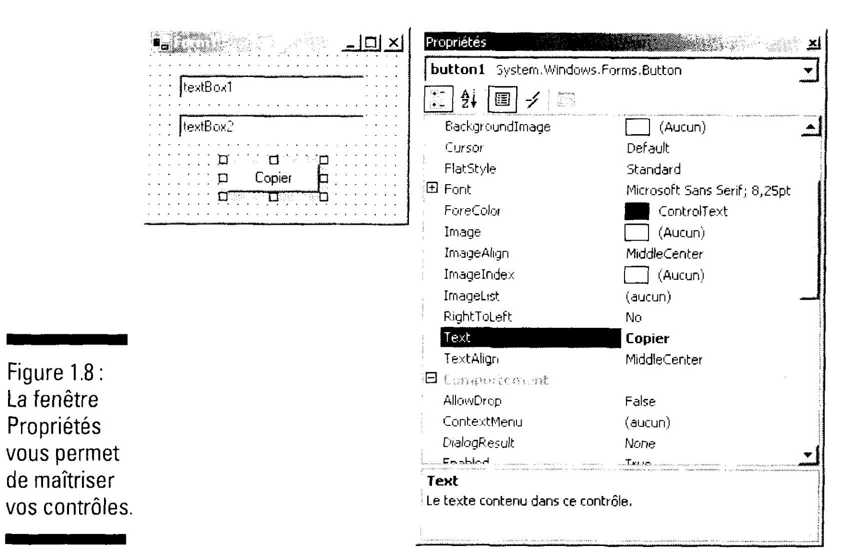


Figure 1.8 : La fenêtre Propriétés vous permet de maîtriser vos contrôles.

Vous pouvez utiliser la propriété Text d'un contrôle zone de texte pour en changer le contenu initial. Pour les deux zones de texte de notre exemple, j'ai défini cette propriété comme "Tapez quelque chose ici" et "Le programme copie ici ce que vous avez tapé", afin que l'utilisateur sache quoi faire après avoir lancé le programme.

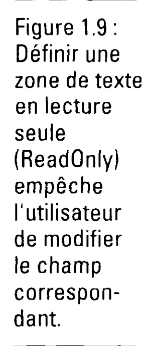
De même, la propriété Text du formulaire lui-même correspond au texte qui apparaît dans sa barre de titre, et vous pouvez la changer. Cliquez dans un endroit quelconque dans le formulaire, tapez ce qui vous convient dans la propriété Text, et appuyez sur Entrée. J'ai fait apparaître dans la barre de titre "Programme qui copie du texte."

4. **Sélectionnez la zone de texte du bas, et faites défiler ses propriétés de comportement pour faire apparaître la propriété ReadOnly (lecture seule). Définissez cette propriété comme vraie en cliquant dessus et en sélectionnant True dans le menu déroulant qui apparaît, comme le montre la Figure 1.9.**

5. **Dans la barre d'outils de Visual Studio, cliquez sur le bouton Enregistrer pour enregistrer votre travail.**



Pendant que vous travaillez, pensez à cliquer régulièrement sur le bouton Enregistrer pour être sûr de perdre le moins de choses possible en cas d'incident.



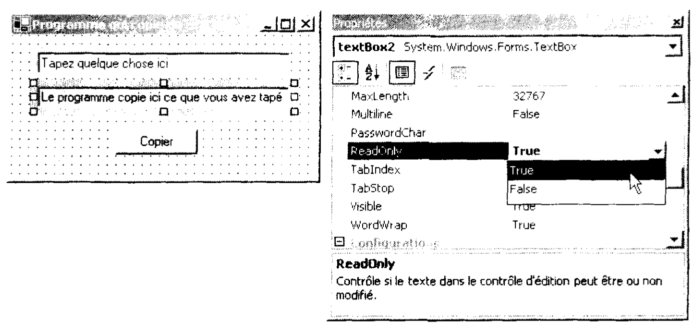


Figure 1.9 : Définir une zone de texte en lecture seule (ReadOnly) empêche l'utilisateur de modifier le champ correspondant.

Générer l'application

Pour générer l'application, sélectionnez Générer/Générer. Cette action génère une nouvelle application Windows avec le formulaire que vous venez de créer. Si elle n'était pas déjà ouverte, la fenêtre Sortie apparaît, dans laquelle vous voyez défiler le flot de messages dont le dernier doit être `Génération : 1 a réussi, 0 a échoué, 0 a été ignoré.`

Vous pouvez maintenant exécuter le programme en sélectionnant Déboguer/Exécuter sans débogage. Le programme ouvre un formulaire conforme à celui que vous venez de créer, comme le montre la Figure 1.10. Vous pouvez taper ce que vous voulez dans la zone de texte du haut, mais vous ne pouvez rien entrer dans celle du bas (à moins d'avoir oublié de définir comme vraie la propriété ReadOnly).

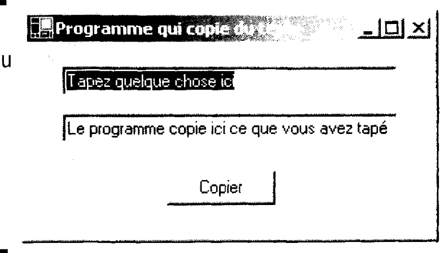


Figure 1.10 : La fenêtre du programme est le formulaire que vous venez de créer.

Faisons-lui faire quelque chose

Ce programme se présente bien, mais il ne fait rien. Si vous cliquez sur le bouton Copier, rien ne se passe. Jusqu'ici, vous n'avez fait que définir les propriétés d'apparence – celles qui définissent l'apparence des contrôles. Il vous faut maintenant mettre dans le bouton Copier l'astuce qui va lui faire effectivement copier le texte de la source à la cible :

1. **Dans le Concepteur de formulaires, sélectionnez le bouton Copier.**

2. **Dans la fenêtre Propriétés, cliquez sur le bouton contenant un éclair, au-dessus de la liste des propriétés, pour ouvrir un nouvel ensemble des propriétés.**

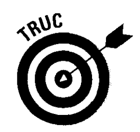
   Ce sont les *événements*. Ils définissent ce que fait un contrôle au cours de l'exécution du programme.

   Vous devez définir l'événement Click. Comme son nom l'indique, il définit ce que fait le bouton lorsque l'utilisateur clique dessus.

3. **Double-cliquez sur l'événement Click et voyez l'écran se transformer.**

   La fenêtre de conception est l'une des deux manières de voir votre application. L'autre est la fenêtre de code qui montre le code source C# que le Concepteur de formulaires a construit pour vous automatiquement. Visual Studio sait qu'il vous faut entrer un peu de code C# afin que votre programme fasse ce que vous attendez de lui.

   Lorsque vous double-cliquez sur la propriété Click, Visual Studio affiche la fenêtre Code et crée une nouvelle *méthode*, à laquelle il donne le nom descriptif `button1_Click()`. Lorsque l'utilisateur clique sur le bouton Copier, cette méthode effectue le transfert du texte de `textBox1`, la source, à `textBox2`, la cible.

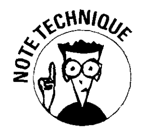


   Pour le moment, ne vous inquiétez pas de ce qu'est une méthode. J'en donnerai la description au Chapitre 8.

   Cette méthode copie simplement la propriété Text de `textBox1` dans la propriété Text de `textBox2`.

4. **Dans la méthode `button1_Click()`, ajoutez simplement la ligne de code suivante :**

```
textBox2.Text = textBox1.Text;
```



Remarquez que C# essaie de vous faciliter la tâche de saisie du code. La Figure 1.11 montre l'affichage au moment où je tape le nom de la dernière propriété Text de la ligne ci-dessus. La liste déroulante des propriétés de la zone de texte correspondante apparaît, vous offrant un aide-mémoire des propriétés disponibles, avec une info-bulle qui vous dit de quoi il s'agit. Cette fonction vous permet de compléter automatiquement ce que vous tapez, et c'est une aide précieuse au cours de la programmation (pour que ça marche, ne faites pas de fautes de frappe dans ce qui précède – attention aux majuscules et minuscules).

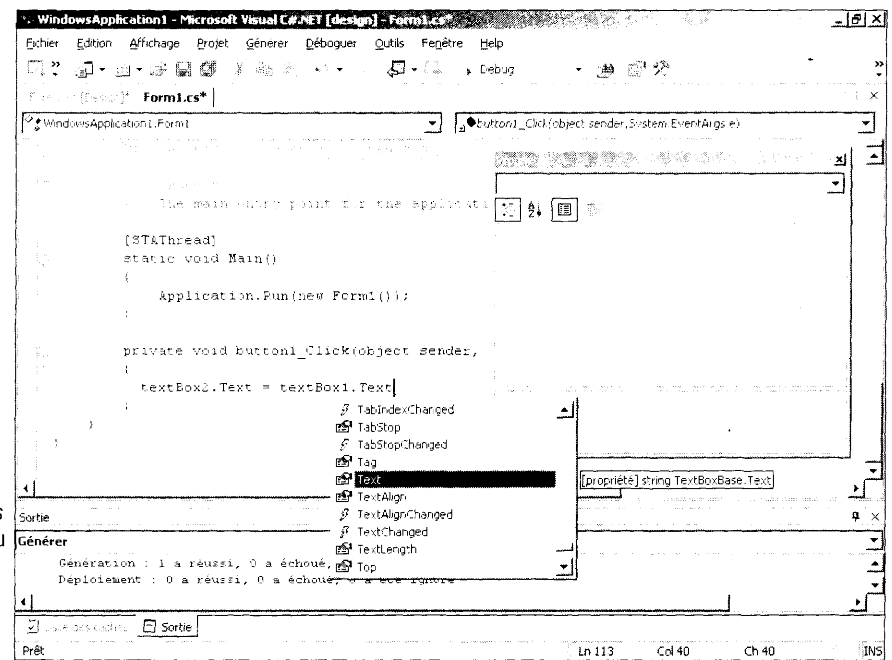


Figure 1.11 : La fonction de suggestion automatique affiche les noms des propriétés au fur et à mesure que vous tapez.

5. **Sélectionnez Générer/Générer pour ajouter au programme la nouvelle méthode de clic.**

Essayer le produit final

Pour exécuter encore une fois le programme, sélectionnez Déboguer/ Exécuter sans débogage. Tapez un peu de texte dans la zone de texte source, et cliquez sur le bouton Copier. Le texte est aussitôt copié dans la

zone de texte cible, comme le montre la Figure 1.12. Vous pouvez joyeusement répéter le processus autant que vous voulez avec tout ce qui vous passe par la tête, jusqu'à ce que l'épuisement vous submerge.

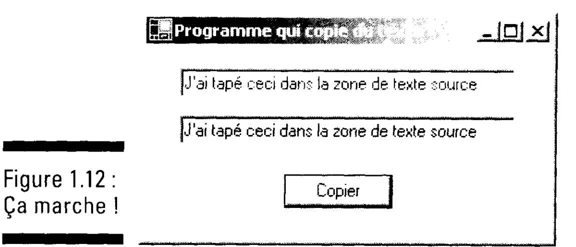
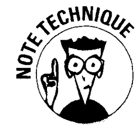


Figure 1.12 : Ça marche !

En considérant le processus de création, vous serez peut-être frappé par son orientation graphique. Faites glisser des contrôles, déposez-les où vous voulez dans le formulaire, définissez leurs propriétés, et vous y êtes. Il vous a suffi d'écrire une seule ligne de code C#, et même ça n'était pas bien difficile.



On pourrait objecter que ce programme ne fait pas grand-chose, mais je ne suis pas d'accord. Consultez simplement un manuel de programmation des débuts de Windows, à l'époque où les assistants d'application n'existaient pas encore, et vous verrez combien d'heures de programmation il fallait pour réaliser une application aussi simple que celle-ci.

Programmeurs Visual Basic, attention !

Ceux d'entre vous qui sont des programmeurs Visual Basic ont peut-être une impression de déjà vu. En fait, le Concepteur de formulaires fonctionne assez largement comme les dernières versions de l'environnement Visual Basic, mais il est beaucoup plus puissant (en tout cas, par rapport aux versions antérieures à Visual Basic .NET). Le langage C# en lui-même est d'ailleurs plus puissant que le précédent Visual Basic. La bibliothèque de routines .NET est plus puissante que l'ancienne bibliothèque de Visual Basic. Enfin, .NET prend en compte le développement distribué et en différents langages, ce que ne faisait pas Visual Basic. En dehors de tout cela, je dirais qu'ils sont à peu près identiques.

Chapitre 2

Créer votre première application console en C#

Dans ce chapitre :

Créer une application console plus maniable.

Examiner le modèle d'application console.

Explorer les différentes parties du modèle.

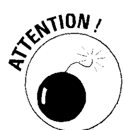
Même le programme Windows le plus élémentaire peut être décourageant pour le programmeur C# débutant. Si vous ne me croyez pas, allez simplement voir le Chapitre 1. Un programme du type que l'on appelle *application console* génère significativement moins de code C# et est beaucoup plus facile à comprendre.

Dans ce chapitre, vous allez utiliser Visual Studio afin de créer un modèle d'application console, que vous allez ensuite simplifier un peu manuellement. Vous pourrez utiliser le résultat comme modèle pour bon nombre des programmes que je présente dans ce livre.



Le principal but de ce livre (en tout cas des premières parties) est de vous aider à comprendre C#. Pour faire en C# un jeu qui aura un succès mondial, il faut d'abord que vous connaissiez le langage C#.

Créer un modèle d'application console



Les instructions suivantes concernent Visual Studio. Si vous utilisez un autre environnement, c'est à la documentation de celui-ci que vous devez vous référer. Mais quel que soit votre environnement, vous pouvez y taper directement le code C#.

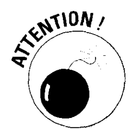
Créer le programme source

Pour créer votre modèle d'application console en C#, suivez ces étapes :

1. **Sélectionnez Fichier/Nouveau/Projet pour créer un nouveau projet.**

   Visual Studio affiche une fenêtre contenant des icônes qui représentent les différents types d'application que vous pouvez créer.

2. **Dans cette fenêtre Nouveau projet, cliquez sur l'icône** `Application console`.



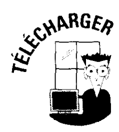
   Faites attention à bien sélectionner le dossier des projets Visual C# dans la fenêtre Nouveau projet. Si vous en sélectionnez un autre par erreur, Visual Studio peut créer une horreur comme une application Visual Basic ou Visual C++.



   Avec Visual Studio, il est nécessaire de créer un projet avant de pouvoir commencer à entrer votre programme C#. Un projet est un peu comme un tiroir dans lequel vous allez entasser tous les fichiers qui constituent votre programme. Lorsque vous demandez au compilateur de générer le programme, il extrait du projet les fichiers dont il a besoin afin de créer le programme à partir de ceux-ci.

   Le nom par défaut de votre première application est `ConsoleApplication1`. L'emplacement par défaut du fichier correspondant est un peu trop en profondeur à mon goût dans le dossier Mes documents. Puisque je suis un peu difficile (ou peut-être parce que j'écris un livre), je préfère classer mes programmes où je veux, et pas nécessairement là où Visual Studio veut les mettre.

3. **Pour changer le dossier par défaut de vos programmes, cliquez sur le bouton Parcourir et naviguez jusqu'à l'endroit voulu.**



   Pour le dossier dans lequel je place tous mes programmes, j'ai choisi le nom Programmes C#.

4. **Dans le champ Nom, entrez le nom que vous voulez donner au projet que vous créez. Pour ce premier programme, nous allons nous en tenir à la tradition avec** `Hello`**.**

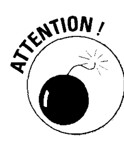
5. **Cliquez sur OK.**

   Après quelques instants de travail, Visual Studio génère un fichier nommé `Class1.cs`. (Si vous regardez dans la fenêtre nommée `Explorateur de solutions`, vous y verrez plusieurs autres fichiers. Vous pouvez les ignorer pour le moment.)

Le contenu de votre première application console apparaît ainsi (les commentaires en anglais seront peut-être en français dans la version que vous utiliserez) :

```
using System;

namespace Hello
{
  /// <summary>
  /// Summary description for Class1.
  /// </summary>
  class Class1
  {
    static void Main(string[] args)
    {
      //
      // TODO: Add code to start application here
      //
    }
  }
}
```



Par rapport au programme exécutable Windows que vous avez créé au Chapitre 1, Visual Studio a inversé les lignes `using System` et `namespace Hello`. C# accepte ces commandes dans un ordre comme dans l'autre (il y a une différence, mais elle est subtile et bien en dehors du cadre de ce chapitre).

Tester le résultat

Sélectionnez Générer/Générer pour faire de votre programme C# un programme exécutable.

Visual Studio répond par le message suivant :

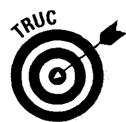
```
- Début de la génération: Projet: Hello, Configuration: Debug .NET -

Préparation des ressources...
Mise à jour des références...
Compilation principale en cours...

Génération terminée -- 0 erreur, 0 avertissement
Génération d'assemblys satellites en cours...

---------------------- Terminé ----------------------
    Génération : 1 a réussi, 0 a échoué, 0 a été ignoré
```

Dans lequel le point important est `1 a réussi`.

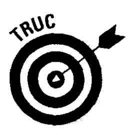


Une règle générale de la programmation est que "a réussi" veut dire "ça va", et "a échoué" veut dire "ça ne va pas".

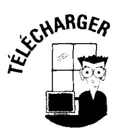
Pour exécuter le programme, sélectionnez Déboguer/Démarrer. Le programme se termine immédiatement. Apparemment, il n'a rien fait du tout, et en fait c'est bien le cas. Le modèle n'est rien d'autre qu'une coquille vide.

Créer votre première véritable application console

Modifiez maintenant le fichier du modèle `Class1.cs`, conformément à ce qui suit.



Ne vous préoccupez pas d'entrer un ou deux espaces ou une ou deux lignes blanches. En revanche, respectez les majuscules et les minuscules.




```
using System;
namespace Hello

{
  public class Class1
  {
    // C'est ici que commence le programme
    static void Main(string[] args)
    {
```

```
            // Demande son nom à l'utilisateur
            Console.WriteLine("Entrez votre nom:");

            // Lit le nom entré par l'utilisateur
            string sName = Console.ReadLine();

            // Accueille l'utilisateur par son nom
            Console.WriteLine("Hello, " + sName);

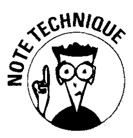
            // Attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
        }
    }
}
```

Sélectionnez Générer/Générer pour faire de cette nouvelle version de `Class1.cs` le programme `Class1.exe`.

Dans la fenêtre Visual Studio .NET, cliquez sur Déboguer/Démarrer. Le programme vous demande immédiatement votre nom. Tapez votre nom et appuyez sur la touche Entrée. Le programme répond :

```
Entrez votre nom:
Hildegarde
Hello, Hildegarde
Appuyez sur Entrée pour terminer...
```

Le programme répond par le mot "Hello", suivi par le nom que vous avez entré. Puis il attend que vous appuyiez sur la touche Entrée pour rendre son dernier soupir.



Vous pouvez aussi exécuter votre programme à partir de la ligne de commande DOS. Ouvrez une fenêtre DOS. Tapez **cd \Programmes C#\Hello\bin\Debug**. Puis, pour exécuter le programme, tapez **Hello** et appuyez sur Entrée. Le résultat doit être le même. Vous pouvez aussi naviguer jusqu'au dossier `\Programmes C#\Hello\bin\Debug` dans l'Explorateur Windows et double-cliquer sur le fichier `Hello.exe`.

Examinons ce programme

Dans les sections qui suivent, nous allons prendre à part l'une après l'autre chaque partie de cette application console en C# afin d'en comprendre le fonctionnement.

Le cadre de travail du programme

Pour toute application console, le cadre de travail de base commence par

```
using System;

namespace Hello
{
  public class Class1
  {
    // C'est ici que commence le programme
    public static void Main(string[] args)
    {
       // Le code sera placé ici
    }
  }
}
```

L'exécution proprement dite du programme commence juste après l'instruction qui contient `Main` et se termine à la parenthèse fermante qui suit `Main`. Je vous expliquerai en temps utile le sens de ces instructions. Je ne peux pas en dire plus pour le moment.



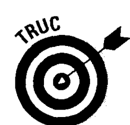
L'instruction `using System` peut venir juste avant ou juste après l'instruction `namespace Hello {`. L'ordre dans lequel elles se présentent n'a pas d'importance.

Les commentaires

Ce modèle contient déjà un certain nombre de lignes, et j'en ai ajouté plusieurs autres, comme ici :

```
// C'est ici que commence le programme
public static void Main(string[] args)
```

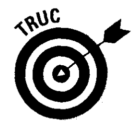
Dans cet exemple, C# ignore la première ligne. C'est ce que l'on appelle un commentaire.



Toute ligne commençant par //ou par /// est une ligne de texte libre qui sera ignorée par C#. Pour le moment, vous pouvez considérer // et /// comme équivalents.

Pourquoi mettre dans un programme des lignes destinées à être ignorées ?

Un programme, même un programme C#, n'est pas facile à comprendre. Souvenez-vous qu'un langage de programmation est un compromis entre ce que comprennent les ordinateurs et ce que comprennent les êtres humains. Introduire des commentaires permet d'expliquer les instructions C#. Ceux-ci peuvent vous aider à écrire le code, et ils seront particulièrement utiles au malchanceux qui devra reprendre votre programme un an plus tard et reconstituer votre logique. Ajouter des explications facilite beaucoup les choses.



N'hésitez pas à faire des commentaires, et faites-en le plus tôt possible. Ils vous aideront, ainsi que les autres programmeurs concernés, à vous rappeler ce que vous avez voulu faire en écrivant ces instructions.

La substance du programme

Le cœur de ce programme se trouve dans le bloc de code délimité par l'instruction `Main ( )` :

```
// Demande son nom à l'utilisateur
Console.WriteLine("Entrez votre nom:");

// Lit le nom entré par l'utilisateur
string sName = Console.ReadLine();

// Accueille l'utilisateur par son nom
Console.WriteLine("Hello, " + sName);
```

L'exécution du programme commence par la première instruction C# : `Console.WriteLine`. Celle-ci écrit dans la console la chaîne de caractères `Entrez votre nom:`.

L'instruction suivante lit la réponse entrée par l'utilisateur et la stocke dans une "boîte de travail" nommée `sName` (j'en dirai plus sur ces emplacements de stockage au Chapitre 3). La troisième instruction combine la chaîne `Hello` et le nom entré par l'utilisateur, et envoie l'ensemble sur la console.

Les trois dernières lignes attendent que l'utilisateur appuie sur la touche Entrée avant de poursuivre. Elles assurent ainsi que l'utilisateur a le temps de lire ce que le programme vient d'afficher :

```
// Attend confirmation de l'utilisateur
Console.WriteLine("Appuyez sur Entrée pour terminer...");
Console.Read();
```

Cette étape peut être importante selon votre environnement et selon la manière dont vous exécutez le programme. Dans Visual Studio, vous avez deux manières d'exécuter un programme. Si vous utilisez la commande Déboguer/Démarrer, Visual Studio ferme la fenêtre de résultats dès que le programme se termine. C'est la même chose qui se produit lorsque vous exécutez le programme en double-cliquant sur l'icône du fichier exécutable dans l'Explorateur Windows.



Quelle que soit la manière dont vous exécutez le programme, attendre que l'utilisateur appuie sur la touche Entrée avant de quitter résout tous les problèmes.

Deuxième partie

Programmation élémentaire en C#



"Excusez-moi. Y a-t-il quelqu'un ici qui ne soit PAS en train de parler de C# ?"

Dans cette partie...

Les programmes les plus récents de commerce électronique, de business to business et de dot.com en tout genre utilisent les mêmes éléments de base que le plus simple programme de conversion de température. Cette partie présente les bases de la création de variables, de l'exécution d'opérations arithmétiques, et de la maîtrise du cheminement de l'exécution d'un programme.

Chapitre 3

Déclarer des variables de type valeur

Dans ce chapitre :

- Créer un emplacement de stockage : la variable en C#.
- Utiliser des entiers.
- Traiter des valeurs fractionnelles.
- Déclarer des variables d'autres types.
- Traiter des constantes numériques.
- Changer de type.

La plus fondamentale de toutes les notions de la programmation est celle de variable. Une variable C# est comme une petite boîte dans laquelle vous pouvez stocker des choses, en particulier des nombres, pour vous en servir ensuite.

Le terme *variable* est emprunté au monde des mathématiques. Par exemple :

```
n = 1
```

signifie qu'à partir du moment où on a écrit cela, on peut utiliser le terme *n* quand on veut dire 1, aussi longtemps que l'on n'aura pas attribué un autre sens à *n* (un nombre, une équation, un concept ou autre).

Dans le monde de la programmation, la signification du mot variable n'est guère différente. Lorsqu'un programmeur C# écrit :

```
int n;
n = 1;
```

Ces instructions définissent un "élément" n, et lui assignent la valeur 1. À partir de ce point dans le programme, la variable n a la valeur 1 jusqu'à ce que le programmeur change cette valeur pour un autre nombre.

Malheureusement pour les programmeurs, C# impose plusieurs limitations aux variables – limitations dont les mathématiciens n'ont pas à se soucier (sauf ceux qui s'aventurent à lire ce livre).

Déclarer une variable

Quand un mathématicien dit : "n égale 1", cela signifie que le terme *n* est équivalent à 1, raisonnement que vous pouvez trouver étrange. Le mathématicien est libre d'introduire des variables au gré de sa fantaisie. Par exemple :

$$x = y^2 + 2y + y$$
$$\text{si } k = y + 1 \text{ alors}$$
$$x = k^2$$

Ici, le mathématicien a écrit une équation quadratique. Peut-être les variables *x* et *y* ont-elles déjà été définies quelque part. Toutefois, lorsqu'il voit apparaître une nouvelle variable, *k*, le programme tombe des nues. Dans cet exemple, *k* ne signifie pas essentiellement qu'il a la valeur de *y* plus 1, mais qu'il représente le concept de *y* plus 1. C'est une sorte de raccourci. Jetez un coup d'œil à n'importe quel manuel de mathématiques, et vous verrez ce que je veux dire. Je dis bien : *jetez un coup d'œil.* Vous êtes ici pour lire mon livre et pas un autre.

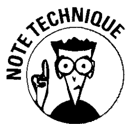
Un programmeur doit être précis dans la terminologie qu'il utilise. Par exemple, il peut écrire le code suivant :

```
int n;
n = 1;
```

La première ligne signifie : "Creuser un petit espace de stockage dans la mémoire de l'ordinateur, et lui assigner le nom *n*." Cette étape revient à ajouter un dossier dans une armoire à dossiers suspendus et à écrire *n* sur son étiquette. La deuxième ligne signifie : "Stocker la valeur 1 dans la variable *n*, en remplaçant par cette valeur tout ce que la variable pouvait contenir auparavant." Dans une armoire à dossiers suspendus, l'équivalent serait : "Ouvrir le dossier *n*, enlever tout ce qu'il contient, et mettre 1 à la place."



Le symbole = est appelé *opérateur d'assignation*. Je dis bien le "symbole" et non le "signe" ou autre terme plus ou moins vague.



Le mathématicien dit : "n égale 1." Le programmeur C# le dit d'une manière plus précise : "Stocker la valeur 1 dans la variable `n`." (Pensez à l'armoire à dossiers suspendus, et vous verrez que c'est préférable.) Les opérateurs C# disent à l'ordinateur ce que vous voulez faire. Autrement dit, les opérateurs sont des verbes et non des descripteurs. L'opérateur d'assignation prend la valeur qui est à sa droite et la stocke dans la variable qui est à sa gauche.

Qu'est-ce qu'un `int` *?*

Les mathématiciens manipulent des concepts. Ils peuvent créer des variables quand ça leur chante, et une même variable peut revêtir différentes significations dans la même équation. Au mieux, un mathématicien considère une variable comme une valeur sans forme fixe, au pire, comme un vague concept. Ne riez pas, c'est probablement de la même manière que vous voyez les choses.

Si le mathématicien écrit :

```
n = 1;
n = 1.1;
n = Maison
n = "Les Martiens sont parmi nous"
```

Chacune de ces lignes associe la variable *n* à une chose différente, et le mathématicien n'y pense même pas. Je n'y pense pas beaucoup moi-même, sauf pour la dernière ligne.

C# est loin d'offrir une telle souplesse. En C#, chaque variable possède un type fixe. Lorsque vous choisissez un nouveau dossier suspendu pour

votre armoire, vous devez en prendre un de la taille qui convient. Si vous avez choisi un dossier suspendu "de type entier", vous ne pouvez pas espérer y mettre la carte de France.

Pour l'exemple de la section précédente, vous allez choisir un dossier suspendu conçu dans le but de contenir un nombre entier : ce que C# appelle une variable de type `int` (integer). Les entiers sont les nombres comme 0, 1, 2, 3, et ainsi de suite, plus les nombres négatifs, -1, -2, -3, et ainsi de suite.



Avant de pouvoir utiliser une variable, vous devez la déclarer. Une fois que vous avez déclaré une variable de type `int`, elle peut contenir et régurgiter des valeurs entières, comme le montre l'exemple suivant :

```
// Déclare une variable entière n
int n;
// Déclare une variable entière m et l'initialise
// avec la valeur 2
int m = 2;
// Assigne à la variable n la valeur stockée dans m
n = m;
```

La première ligne après le commentaire est une déclaration qui crée une zone de stockage, `n`, faite pour contenir une valeur entière. Aussi longtemps qu'il ne lui est pas assigné une valeur, la valeur initiale de `n` n'est pas spécifiée. La deuxième déclaration crée une variable entière `m`, avec 2 pour valeur initiale.



Le terme *initialiser* signifie assigner une valeur initiale. Initialiser une variable, c'est lui assigner une valeur pour la première fois. Tant qu'une variable n'a pas été initialisée, on n'en connaît pas la valeur.

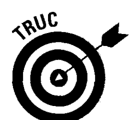
La dernière instruction de cet exemple assigne à la variable `n` la valeur stockée dans m, qui est 2. La variable `n` continue à contenir la valeur 2 jusqu'à ce qu'une nouvelle valeur lui soit assignée.

Les règles de déclaration de variable

Vous pouvez initialiser une variable dans la déclaration elle-même :

```
// Déclare une nouvelle variable int
// et lui donne 1 comme valeur initiale
int o = 1;
```

Ce qui revient à mettre la valeur 1 dans votre nouveau dossier suspendu au moment où vous le mettez dans l'armoire, plutôt que de le mettre d'abord pour le rouvrir ensuite et y mettre la valeur.



Il vaut mieux initialiser une variable dans la déclaration. Dans la plupart des cas, mais pas tous, C# initialisera la variable pour vous, mais il vaut mieux ne pas dépendre de lui pour cela.

Vous pouvez déclarer une variable n'importe où (en fait, presque n'importe où) dans un programme, mais vous ne pouvez pas utiliser une variable avant de l'avoir déclarée et de lui avoir donné une valeur. Les deux instructions suivantes sont donc illicites :

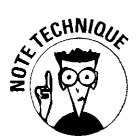
```
// Ce qui suit est illicite car m n'a pas reçu
// une valeur avant d'être utilisée
int m;
n = m;
// Ce qui suit est illicite car p n'a pas été
// déclarée avant d'être utilisée
p = 2;
int p;
```

Enfin, vous ne pouvez pas déclarer deux fois la même variable.

Variations sur un thème : des `int` *de différents types*

La plupart des variables simples sont de type `int`. C# en offre un certain nombre de variantes pour quelques occasions particulières.

Toutes les variables de type `int` sont limitées aux nombres entiers. Le type `int` souffre aussi d'autres limitations. Par exemple, une variable de type `int` ne peut stocker de valeur qu'à l'intérieur de l'étendue -2 milliards à 2 milliards.

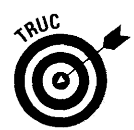


L'étendue exacte est de -2 147 483 648 à 2 147 483 647.



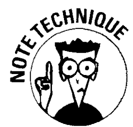
Deux milliards de centimètres représentent à peu près la moitié de la circonférence de la Terre.

Au cas où 2 milliards ne vous suffirait pas, C# offre un type d'entier nommé `long` (abréviation de `long int`) qui peut contenir un nombre entier beaucoup plus long. Le seul inconvénient du type `long` est qu'il occupe plus de place dans votre armoire : un entier de type `long` consomme huit octets (64 bits), soit deux fois plus qu'un `int` ordinaire.



Autrement dit, un entier `long` occupe deux dossiers d'une capacité d'un `int` dans votre armoire à dossiers suspendus. Comme cette métaphore d'armoire à dossiers suspendus commence à être un peu usée, je parlerai à partir de maintenant en octets.

Un entier `long` représente un nombre entier qui peut aller approximativement de -10^{19} à 10^{19}.



L'étendue exacte d'un entier long est de -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807.

C# offre plusieurs autres types de variable entière montrés par le Tableau 3.1.

Tableau 3.1 : Taille et étendue des types entiers en C#.

Type	Taille (octets)	Étendue	Exemple
sbyte	1	-128 à 127	sbyte sb = 12;
byte	1	0 à 255	byte b = 12;
short	2	-32,768 à 32,767	short sn = 123456;
ushort	2	0 à 65,535	ushort usn = 62345678;
`int`	4	-2 milliards à 2 milliards	`int` n = 1234567890;
uint	4	0 à 4 milliards	uint un = 3234567890U
long	8	-10^{19} à 10^{19} (beaucoup)	long l = 123456789012L
ulong	8	0 à 10^{20}	long ul = 123456789012UL

Comme je l'expliquerai dans la section "Déclarer des constantes numériques", plus loin dans ce chapitre, une valeur fixe telle que 1 a aussi un type. Par défaut, une constante simple comme 1 est considérée de type `int`. Une constante de type autre que `int` doit être marquée par son type. Par exemple, 123U est un entier non signé (unsigned), de type `uint`.

La plupart des variables de type entier sont *signées*, ce qui signifie qu'elles ont un signe (+ ou -), et qu'elles peuvent donc représenter des valeurs négatives. Un entier non signé ne peut représenter que des valeurs positives, avec l'avantage de pouvoir contenir une valeur deux fois plus élevée. Comme vous pouvez le voir dans le Tableau 3.1, les noms de la plupart des types d'entier non signé commencent par `u` (pour *unsigned*), alors que les types signés n'ont généralement pas de préfixe.

Représenter des fractions

Les entiers conviennent très bien pour la plupart des calculs, mais beaucoup font intervenir des fractions, qui ne peuvent être représentées par des nombres entiers. L'équation toute simple qui convertit en degrés Celsius une température exprimée en degrés Fahrenheit met le problème en évidence :

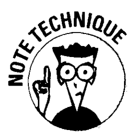
```
// Conversion de la température 41 degrés Fahrenheit
int nFahr = 41;
int nCelsius = (nFahr - 32) * (5 / 9)
```

Cette équation fonctionnera très bien en nombres entiers pour certaines valeurs. Par exemple, 41 degrés Fahrenheit équivaut à 5 degrés Celsius.

Essayons une autre valeur : 100 degrés Fahrenheit. Selon notre équation, 100-32 = 68 ; 68 fois 5/9 = 37. C'est faux : la bonne réponse est 37,78. Mais cette réponse est encore fausse car le résultat exact est 37,777... avec des 7 jusqu'à l'infini.

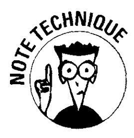


Une variable `int` ne peut représenter que des nombres entiers. L'équivalent entier de 37,78 est 37. Cette manière d'escamoter la partie décimale d'un nombre pour le faire tenir dans une variable entière s'appelle *tronquer.*

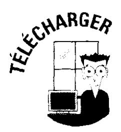


Tronquer n'est pas la même chose qu'arrondir. Tronquer consiste à supprimer la partie décimale, alors qu'arrondir consiste à prendre la valeur entière la plus proche. Ainsi, tronquer 1,9 donne 1. Arrondir 1,9 donne la valeur 2.

Pour une température, 37 peut être une approximation satisfaisante. On ne va pas mettre une chemise à manches longues à 37 degrés et une chemise à manches courtes à 37,7 degrés. Mais pour bien des applications, sinon presque toutes, il est inacceptable de tronquer un nombre.



En fait, c'est plus compliqué que ça. Une variable de type `int` ne peut pas stoker la fraction 5/9 qui correspond toujours pour elle à la valeur 0. Il en résulte que l'équation de notre exemple ci-dessus calcule la valeur 0 pour `nCelsius` pour toutes les valeurs de `nFahr`. On est bien obligé d'admettre que c'est inacceptable.



Sur le site Web, le dossier `ConvertTemperatureWithRoundOff`, contient un programme de conversion de température avec des variables de type `int`. À ce stade, vous n'en comprendrez peut-être pas tous les détails, mais vous pouvez jeter un coup d'œil aux équations et exécuter le programme `Class1.exe` pour voir les résultats qu'il produit.

Utiliser des variables en virgule flottante

Les limitations d'une variable de type `int` sont inacceptables pour la plupart des applications. L'étendue n'est généralement pas le problème : un entier long de 64 bits à des chances d'être suffisant pour tout le monde. Ce qui est difficile à avaler, c'est de n'avoir droit qu'aux nombres entiers.

Dans bien des cas, vous aurez besoin de nombres dont la partie décimale n'est pas nulle. Les mathématiciens les appellent les *nombres réels*. J'ai toujours trouvé ça ridicule. Y a-t-il des nombres entiers qui soient irréels ?



Remarquez que j'ai dit qu'un nombre réel peut avoir une partie décimale non nulle, mais ce n'est pas obligatoire. Autrement dit, 1,5 est un nombre réel, mais 1,0 également. Par exemple, 1,0 + 0,1 égale 1,1. Un nombre réel plus un nombre réel donne un nombre réel. Gardez cela en tête pour la suite de ce chapitre.

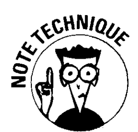
Heureusement, C# connaît les nombres réels. Il y en a de deux sortes : décimaux et à virgule flottante. Le type le plus courant est à virgule flottante. Je décrirai le type `decimal` un peu plus loin dans ce chapitre.

Déclarer une variable à virgule flottante

Une variable en virgule flottante est de type `float`, et vous pouvez la déclarer de la façon suivante :

```
float f = 1.0;
```

Une fois déclarée comme `float`, la variable `f` est de type `float` pour toutes les instructions qui vont s'y appliquer.



Un nombre à virgule flottante doit son nom au fait que la virgule décimale y est autorisée à "flotter" de gauche à droite au lieu de se trouver à un emplacement fixe. Il permet donc de représenter aussi bien 10,0 que 1,00 ou 0,100, ou tout ce que l'on peut imaginer.

Le Tableau 3.2 donne l'ensemble des types de variable en virgule flottante. Tous ces types sont signés, ce qui veut dire qu'une variable en virgule flottante peut recevoir une valeur négative aussi bien que positive.

Tableau 3.2 : Taille et étendue des types de variable en virgule flottante.

Type	Taille [octets]	Étendue	Précision	Exemple
float	8	$1.5 * 10^{-45}$ à $3.4 * 10^{38}$	6 - 7 chiffres	float f = 1.2F;
double	16	$5.0 * 10^{-324}$ à $1.7 * 10^{308}$	15 - 16 chiffres	double d = 1.2;

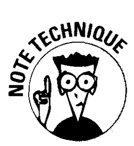


Le type par défaut pour une variable en virgule flottante est `double` et non `float`.

Dans le Tableau 3.2, la colonne Précision contient le nombre de chiffres exacts pour chaque type de variable à virgule flottante. Par exemple, la fraction 5/9 vaut exactement 0,555... avec une infinité de 5. Mais une variable de type `float` contient un certain nombre de chiffres exacts, ce qui veut dire que les chiffres qui se trouvent au-delà du sixième ne le sont pas nécessairement. Aussi, exprimé dans le type `float`, 5/9 pourrait très bien apparaître comme ceci :

```
0.5555551457382
```

Vous savez donc que tous les chiffres apparaissant après le sixième 5 ne peuvent pas être considérés comme exacts.

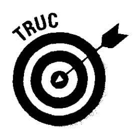


Une variable de type `float` possède en fait 6,5 chiffres exacts. Cette valeur étrange vient du fait que la précision en virgule flottante est donnée par un calcul qui fait intervenir 10 puissance un logarithme en base 2. Voulez-vous vraiment en savoir plus ?

Avec une variable de type `double`, la même fraction 5/9 pourra apparaître de cette façon :

```
0.55555555555555557823
```

Le type double possède quant à lui entre 15 et 16 chiffres exacts.

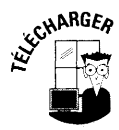


Comme en C# une variable à virgule flottante est par défaut en double précision (le type `double`), vous pouvez utiliser ce type, à moins d'avoir une raison particulière de ne pas le faire. Toutefois, qu'il utilise le type `double` ou le type `float`, on dira toujours d'un programme qu'il travaille en virgule flottante.

Convertissons encore quelques températures

Voici la formule qui permet de convertir en degrés Celsius une température en degrés Fahrenheit en utilisant des variables en virgule flottante :

```
double dCelsius = (dFahr - 32.0) * (5.0 / 9.0)
```



Le site Web contient une version en virgule flottante, nommée `ConvertTemperatureWithFloat`, du programme de conversion de température.

L'exemple suivant montre le résultat de l'exécution du programme `ConvertTemperatureWithFloat`, utilisant des variables de type `double` :

```
Entrez la température en degrés Fahrenheit:100
Température en degrés Celsius = 37.7777777777777779
Appuyez sur Entrée pour quitter le programme...
```

C'est mieux que les problèmes de robinet que l'on apprenait à résoudre à école primaire.

Quelques limitations des variables en virgule flottante

Vous pouvez être tenté d'utiliser tout le temps des variables à virgule flottante parce qu'elles résolvent le problème des nombres tronqués. Il

est vrai qu'elles utilisent plus de mémoire, mais de nos jours la mémoire ne coûte pas cher. Alors, pourquoi pas ? Mais les variables à virgule flottante ont aussi des limitations.

Utiliser une variable comme compteur

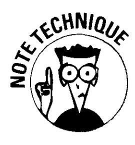
Vous ne pouvez pas utiliser une variable à virgule flottante comme compteur. En C#, certaines structures ont besoin de compter (comme dans 1, 2, 3, et ainsi de suite). Nous savons tous que 1,0, 2,0, et 3,0 font aussi bien que 1, 2, 3 pour compter, mais C# ne le sait pas. Comment ferait C# pour savoir si vous voulez dire 10 000 001 ou 10 000 000 ?

Que vous trouviez ou non cet argument convaincant, vous ne pouvez pas utiliser une variable à virgule flottante comme compteur.

Comparer des nombres

Il faut être très prudent quand on compare des nombres en virgule flottante. Par exemple, 12,5 peut être représenté comme 12,500001. La plupart des gens ne se préoccupent pas de la précision de ce petit 1 supplémentaire, mais un ordinateur prend les choses de façon extrêmement littérale. Pour C#, 12,500000 et 12,500001 ne sont pas du tout la même chose.

Aussi, si vous ajoutez 1,1 à ce que vous voyez comme 1,1, vous ne pouvez pas savoir a priori si le résultat est 2,2 ou 2,200001. Et si vous demandez "`dDoubleVariable` est-elle égale à 2,2 ?", vous n'aurez pas forcément le résultat que vous attendez. En général, vous allez devoir vous en remettre à une comparaison un peu factice comme celle-ci : "La valeur absolue de la différence entre `dDoubleVariable` et 2,2 est-elle inférieure à 0,000001 ?"

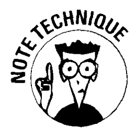


Le processeur Pentium a une astuce pour rendre ce problème moins gênant : il effectue les calculs en virgule flottante dans un format particulièrement long, c'est-à-dire qu'il utilise 80 bits au lieu de 64. Quand on arrondit un nombre en virgule flottante de 80 bits pour en faire une variable de type `float` de 64 bits, on obtient (presque) toujours le résultat attendu, même s'il y avait un ou deux bits erronés dans le nombre de 80 bits.

La vitesse de calcul

Les processeurs de la famille x86 utilisés par les PC un peu anciens fonctionnant sous Windows faisaient les calculs arithmétiques en nombres entiers beaucoup plus vite que les mêmes calculs avec des nombres en virgule flottante. De nos jours, ce serait sortir de ses habitudes pour un programmeur que de limiter son programme à des calculs arithmétiques en nombres entiers.

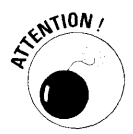
Avec le processeur Pentium III de mon PC, pour un simple test (peut-être trop simple) d'à peu près 300 millions d'additions et de soustractions, le rapport de vitesse a été d'environ 3 à 1. Autrement dit, pour toute addition en type `double`, j'aurais pu faire trois additions en type `int` (les calculs comportant des multiplications et des divisions donneraient peut-être des résultats différents).



J'ai dû écrire mes opérations d'addition et de soustraction de manière à éviter les effets de cache. Le programme et les données étaient mis en cache, mais le compilateur ne pouvait mettre en cache dans les registres du CPU aucun résultat intermédiaire.

Une étendue pas si limitée

Dans le passé, une variable en virgule flottante pouvait posséder une étendue beaucoup plus large qu'une variable d'un type entier. C'est toujours le cas, mais l'étendue du type `long` est assez grande pour rendre la question pratiquement dépourvue d'intérêt.



Même si une variable de type `float` peut représenter un assez grand nombre, le nombre de chiffres exacts est limité. Par exemple, il n'y aura pas de différence entre 123456789F et 123456000F.

Utiliser le type `decimal`, hybride d'entier et de virgule flottante

Comme je l'explique dans les sections précédentes de ce chapitre, les types entiers comme les types en virgule flottante ont chacun leurs inconvénients. Une variable en virgule flottante a des problèmes d'arrondi, ainsi que des limites de précision, alors qu'une variable `int` fait

tout simplement sauter la partie décimale du nombre. Dans certains cas, il vous faudra une variable qui combine le meilleur de ces deux types :

- Comme une variable en virgule flottante, pouvoir représenter une fraction.
- Comme une variable entière, offrir une valeur exacte utilisable dans des calculs. Par exemple, 12,5 est effectivement 12,5, et non 12,500001.

Heureusement, C# offre un tel type de variable, nommé `decimal`. Une variable de ce type peut représenter tout nombre compris entre 10^{-28} et 10^{28} (ça fait beaucoup de zéros). Et elle le fait sans problèmes d'arrondi.

Déclarer une variable de type `decimal`

Une variable de type `decimal` se déclare comme n'importe quelle autre :

```
decimal m1;          // Bien
decimal m2 = 100;    // Mieux
decimal m3 = 100M;   // Encore mieux
```

La déclaration de `m1` définit une variable `m1` sans lui donner une valeur initiale. Tant que vous ne lui aurez pas assigné une valeur, son contenu est indéterminé. C'est sans importance, car C# ne vous laissera pas utiliser cette variable tant que vous ne lui aurez pas donné une valeur.

La seconde déclaration crée une variable `m2` et lui donne 100 comme valeur initiale. Ce qui n'est pas évident, c'est que 100 est en fait de type `int`. C# doit donc convertir cette valeur de type `int` en type `decimal` avant de l'initialiser. Heureusement, C# comprend ce que vous voulez dire et effectue la conversion pour vous.

La déclaration de `m3` est la meilleure des trois. Elle initialise `m3` avec la constante de type `decimal` 100M. La lettre M à la fin du nombre signifie que la constante est de type `decimal`. (Voyez la section "Déclarer des constantes numériques", plus loin dans ce chapitre.)

Comparer les types decimal*,* int*, et* float

Une variable de type `decimal` semble avoir tous les avantages et aucun des inconvénients des types `int` et `double`. Elle a une très grande étendue, elle n'a pas de problèmes d'arrondi, et 25,0 y est bien 25,0 et non 25,00001.

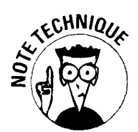
Le type `decimal` a toutefois deux limitations significatives. Tout d'abord, une variable de type `decimal` ne peut pas être utilisée comme compteur, car elle peut contenir une partie décimale. Vous ne pouvez donc pas vous en servir dans une boucle de contrôle de flux, comme je l'explique au Chapitre 5.

Le second inconvénient du type `decimal` est tout aussi sérieux, ou même plus. Les calculs effectués avec des variables de ce type sont significativement plus lents que ceux effectués avec des variables de type `int` ou `float`. Je dis bien "significativement". Mon test simple de 300 millions d'additions et de soustractions a été à peu près cinquante fois plus long qu'avec le type `int`, et je soupçonne que ce rapport serait encore plus défavorable avec des opérations plus complexes. En outre, la plupart des fonctions de calcul, comme les exponentielles ou les fonctions trigonométriques n'admettent pas le type `decimal`.

Il est clair que le type `decimal` convient très bien aux applications comme la comptabilité, pour lesquelles la précision est très importante mais le nombre de calculs relativement réduit.

Soyons logique, examinons le type bool

Enfin, un type de variable logique. Une variable du type booléen `bool` peut prendre deux valeurs : `true` ou `false` (vrai ou faux). Je parle sérieusement : un type de variable rien que pour deux valeurs.



Les programmeurs C et C++ ont l'habitude d'utiliser une variable `int` avec la valeur 0 (zéro) pour signifier `false`, et une valeur autre que zéro pour signifier `true`. Ça ne marche pas en C#.

Une variable `bool` se déclare de la façon suivante :

```
bool variableBool = true;
```

Il n'existe aucun chemin de conversion entre une variable `bool` et tous les autres types. Autrement dit, vous ne pouvez pas convertir directement une variable `bool` en quelque chose d'autre (et même si vous pouviez, vous ne devriez pas, parce que ça n'a aucun sens). En particulier, vous ne pouvez pas transformer une variable `bool` en `int` (par exemple sur la base du principe que `false` devient zéro), ni en `string` (par exemple sur la base du principe que `false` devient "`false`").

Toutefois, une variable de type `bool` joue un rôle important pour forcer l'exécution d'un programme C# à suivre tel ou tel cheminement, comme je l'explique au Chapitre 5.

Un coup d'œil aux types caractère

Un programme qui ne fait rien d'autre que cracher des nombres peut convenir très bien à des mathématiciens, des comptables, des assureurs qui font des statistiques, et des gens qui font des calculs balistiques (ne riez pas, les premiers ordinateurs ont été construits pour générer des tables de trajectoires d'obus à l'usage des artilleurs). Mais pour la plupart des applications, les programmes doivent pouvoir traiter des lettres aussi bien que des nombres.

C# dispose de deux manières différentes de traiter les caractères : à titre individuel (par le type `char`), et sous forme de chaînes (par le type `string`).

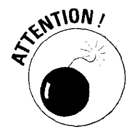
La variable de type `char`

Une variable de type `char` est une boîte qui peut contenir un seul caractère. Une constante caractère apparaît tel un caractère entouré d'apostrophes, comme dans cet exemple :

```
char c = 'a';
```

Vous pouvez y mettre n'importe quel caractère de l'alphabet latin, hébreu, arabe, cyrillique, et bien d'autres. Vous pouvez aussi y mettre des caractères japonais Katakana ou bien des caractères Kanji, chinois ou japonais.

En outre, le type `char` peut être utilisé comme compteur, ce qui veut dire que vous pouvez avoir recours à une variable `char` pour contrôler les structures de boucle que je décrirai au Chapitre 5. Une variable caractère ne peut avoir aucun problème d'arrondi.



Une variable de type char n'est pas associée à une police. Vous pouvez très bien y stocker un caractère Kanji que vous trouvez très beau, mais si vous l'affichez, il ne ressemblera à rien si vous ne le faites pas avec la bonne police.

Types char spéciaux

Certains caractères, selon la police utilisée, ne sont pas imprimables, au sens où vous ne voyez rien si vous les affichez à l'écran ou si vous les imprimez avec votre imprimante. L'exemple le plus banal en est l'espace, représenté par ' '. Il y a aussi des caractères qui n'ont pas d'équivalent sous forme de lettre (par exemple, le caractère de tabulation). Pour représenter ces caractères, C# utilise la barre oblique inverse, comme le montre le Tableau 3.3.

Tableau 3.3 : Caractères spéciaux.

Constante caractère	Valeur
'\n'	nouvelle ligne
'\t'	tabulation
'\0'	caractère null
'\r'	retour chariot
'\\'	barre oblique inverse

Le type string

Le type string est également d'usage courant. Les exemples suivants montrent comment déclarer et initialiser une variable de type string :

```
// déclaration puis initialisation
string someString1;
someString1 = "ceci est une chaîne";
// initialisation avec la déclaration
string someString2 = "ceci est une chaîne";
```

Une constante de type chaîne est une chaîne de caractères entourée de guillemets. Une chaîne peut contenir tous les caractères spéciaux du Tableau 3.3. Une chaîne ne peut pas s'étendre sur plus d'une ligne dans le fichier source C#, mais elle peut contenir le caractère de retour à la ligne, comme le montre l'exemple suivant :

```
// ceci est illicite
string someString = "Ceci est une ligne
et en voilà une autre";
// mais ceci est autorisé
string someString = "Ceci est une ligne\net en voilà une autre";
```

À l'exécution, la dernière ligne de l'exemple ci-dessus écrit les deux membres de phrase sur deux lignes successives :

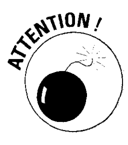
```
Ceci est une ligne
et en voilà une autre
```

Une variable de type `string` ne peut pas être utilisée comme compteur, et ce n'est pas un type contenant une valeur. Il n'existe aucune "chaîne" qui ait une signification intrinsèque pour le processeur. Seul un des opérateurs habituels est utilisable avec un objet de type `string` : l'opérateur `+` effectue la concaténation de deux chaînes en une seule. Par exemple :

```
string s = "Ceci est un membre de phrase"
        + " et en voilà un autre";
```

Ce code place dans la variable `string s` la chaîne suivante :

```
"Ceci est un membre de phrase et en voilà un autre"
```



Encore un mot : une chaîne ne contenant aucun caractère (que l'on écrira "") est une chaîne valide.

Comparer `string` *et* `char`

Bien qu'une chaîne soit constituée de caractères, le type `string` est très différent du type `char`. Naturellement, il existe quelques différences triviales. Un caractère est placé entre apostrophes, comme ceci :

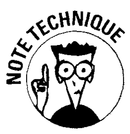
```
'a'
```

alors qu'une chaîne est placée entre guillemets :

```
"ceci est une chaîne"
```

Les règles qui s'appliquent aux chaînes ne sont pas les mêmes que celles qui s'appliquent aux caractères. Pour commencer, une variable `char` ne contient par définition qu'un seul caractère. Le code suivant, par exemple, n'a aucun sens :

```
char c1 = 'a';
char c2 = 'b';
char c3 = c1 + c2
```

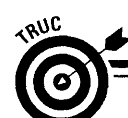


En fait, ce code peut presque se compiler, mais avec une signification complètement différente de l'intention initiale. Ces instructions provoquent la conversion de `c1` en une variable `int` qui reçoit la valeur numérique du caractère initialement contenu dans `c1`, puis la même chose pour `c2`, et finalement l'addition de ces deux entiers. L'erreur se produit lorsque l'on essaie de stocker le résultat dans `c3`. Une valeur numérique de type `int` ne peut pas être stockée dans une variable de type `char`, par définition plus petite. En tout cas, cette opération n'a aucun sens.

Une chaîne, en revanche, peut avoir une longueur quelconque, et la concaténation de deux chaînes a un sens :

```
string s1 = "a";
string s2 = "b";
string s3 = s1 + s2;  //le résultat est "ab"
```

Il y a dans la bibliothèque de C# une collection entière d'opérations sur les chaînes. Je les décrirai au Chapitre 9.



Conventions sur les noms

La programmation est assez difficile sans que les programmeurs la rendent plus difficile encore. Pour rendre votre code source C# plus lisible, adoptez une convention sur les noms, et tenez-vous-y. Idéalement, cette convention doit être aussi proche que possible de celle adoptée par la plupart des programmeurs C#.

La règle générale est que le nom d'un objet autre qu'une variable doit commencer par une lettre majuscule. Choisissez des noms aussi descriptifs que possible, ce qui signifie bien souvent des noms composés de plusieurs mots. Chacun de ces mots doit commencer par une majuscule (sauf le premier dans le cas d'une variable), et ils doivent être collés les uns aux autres, sans être reliés par des tirets de soulignement, comme ceci : `ceciEstUnLongNomDeVariable`

J'ai aussi adopté la règle supplémentaire selon laquelle la première lettre du nom d'une variable en indique le type. La plupart de ces lettres sont assez explicites : `f` pour `float`, `d` pour `double`, `s` pour `string`, et ainsi de suite. La seule qui nécessite un petit effort de mémoire est `n` pour `int`. Il y a toutefois une exception à cette règle : pour des raisons qui remontent au langage FORTRAN des années soixante, les lettres `i`, `j`, et `k` sont couramment utilisés seules comme nom pour les variables de type `int`.

Remarquez que j'ai dit avoir "adopté" cette convention. Ce n'est pas moi qui l'ai inventée, mais quelqu'un qui travaillait chez Microsoft à l'époque du langage C. Comme il était d'origine hongroise, cette convention a reçu le nom de notation hongroise.

La notation hongroise semble ne plus être en faveur. Je continue toutefois à la préférer, car elle me permet de connaître du premier coup d'œil le type de chaque variable dans un programme sans avoir à me référer à la déclaration.

Qu'est-ce qu'un type valeur ?

Toutes les instructions C# doivent être traduites en instructions machine du processeur utilisé. Dans le cas du PC, un processeur Intel. Ces processeurs utilisent aussi la notion de variable. Par exemple, le processeur Intel comporte huit emplacements internes, appelés *registres*, dont chacun peut stocker un `int`. Sans trop entrer dans les détails, je dirai simplement que les types décrits dans ce chapitre, à l'exception de `decimal` et `string`, sont intrinsèques au processeur. Aussi, il existe une instruction machine qui signifie : "Ajouter un `int` à un autre `int`", et une instruction semblable pour ajouter un `double` à un `double`. Comme ces types de variable sont intégrés au fonctionnement du processeur, on les appelle des types *intrinsèques*.

En outre, les types de variable que je décris dans ce chapitre sont de longueur fixe (encore une fois, à l'exception de `string`). Une variable de longueur fixe occupe toujours la même quantité de mémoire. Ainsi, si j'écris l'instruction `a = b`, C# peut transférer la valeur de `b` dans `a` sans prendre

des mesures spéciales pour traiter un type de donnée de longueur variable. Un type de variable qui a cette caractéristique est appelé *type valeur.*



Les types `int`, `double`, `bool`, et leurs dérivés immédiats, comme `int` non signé, sont des types de variable intrinsèques. Les types de variable intrinsèques, ainsi que le type `decimal`, sont aussi appelés types valeur. Le type `string` n'est ni l'un ni l'autre.

Les types définis par le programmeur, que je décris au Chapitre 6, ne sont ni des types valeur, ni des types intrinsèques.

Déclarer des constantes numériques

Bien qu'il y ait très peu d'absolus dans la vie, je vais vous dire quelque chose d'absolu sur C# :



Toute expression a une valeur et un type.

Dans une déclaration comme `int n`, vous pouvez facilement voir que la variable `n` est de type `int`, et vous pouvez raisonnablement supposer que le calcul n + 1 est de type `int`. Mais quel est le type de la constante 1 ?

Le type d'une constante dépend de deux choses : sa valeur, et la présence optionnelle d'une lettre descriptive à la fin de celle-ci. Tout nombre entier inférieur à 2 milliards est supposé être de type `int`. Un nombre entier supérieur à 2 milliards est supposé être de type `long`. Tout nombre en virgule flottante est supposé être de type `double`.

Le Tableau 3.4 présente des constantes déclarées pour être d'un type particulier. La lettre qui sert de descripteur peut être aussi bien en majuscule qu'en minuscule. `1U` et `1u` sont équivalents.

Tableau 3.4 : Constantes déclarées avec leur type.

Constante	Type
1	int
1U	unsigned int
1L	long int
1.0	double

Constante	Type
`1.0F`	`float`
`1M`	`decimal`
`true`	`bool`
`false`	`bool`
`'a'`	`char`
`'\n'`	`char` (caractère de nouvelle ligne)
`'\x123'`	`char` (caractère dont la valeur numérique est hex 123)
`"a string"`	`string`
`""`	`string` (chaîne vide)

Changer de type : le cast

Un être humain ne traite pas de manière différente les différents types de nombre. Par exemple, un individu normal (contrairement à un programmeur C# distingué tel que vous) ne se demande pas si le nombre 1 est signé, non signé, court ou long.

Bien que C# considère ces divers types comme différents, il n'ignore pas qu'il existe une relation entre eux. Par exemple, le code suivant convertit une variable de type `int` en `long` :

```
int nValue = 10;
long lValue;
lValue = nValue;  // ceci fonctionne
```

Une variable de type `int` peut être convertie en `long` car toute valeur `int` peut être stockée dans une variable de type `long` et que l'un comme l'autre type peut être utilisé comme compteur.

Toutefois, une conversion dans la direction opposée peut poser des problèmes. Par exemple, ce qui suit est illicite :

```
long lValue = 10;
int nValue;
nValue = lValue;  // ceci est illicite
```

Certaines valeurs que l'on peut stocker dans une variable de type long sont trop grandes pour une variable int (par exemple, 4 milliards). Dans ce cas, C# génère une erreur, car des informations pourraient être perdues dans la conversion. Ce type de bogue est très difficile à identifier.

Mais si vous savez que la conversion est possible ? Par exemple, bien que lValue soit de type long, peut-être savez-vous que sa valeur ne peut pas dépasser 100 dans ce programme. Dans ce cas, la conversion de la variable de type long lValue en variable nValue de type int ne poserait aucun problème.

Vous pouvez dire à C# que vous savez ce que vous faites en utilisant un *cast* :

```
long lValue = 10;
int nValue;
nValue = (int)lValue;  // maintenant ça marche
```

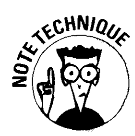
Dans un cast, vous placez entre parenthèses le nom du type que vous voulez obtenir, juste avant le nom de la variable à convertir. Le cast ci-dessus dit : "Convertir en int la valeur de lValue. Je sais ce que je fais."

Un nombre qui peut être utilisé comme compteur peut être converti automatiquement en type float, mais la conversion d'un nombre en virgule flottante en nombre pouvant servir de compteur nécessite un cast :

```
double dValue = 10.0;
long lValue = (long)dValue;
```

Toute conversion de et vers le type decimal nécessite un cast. En fait, tout type numérique peut être converti en n'importe quel autre type numérique par l'application d'un cast.

Ni le type bool ni le type string ne peuvent être convertis directement en un autre type, quel qu'il soit.



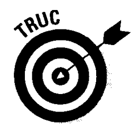
C# comporte des fonctions intégrées qui peuvent convertir un nombre, un caractère ou un type booléen en son équivalent de type string. Par exemple, vous pouvez convertir la valeur de type bool true pour en faire la valeur string "true", mais on ne peut pas considérer cela comme une conversion directe. La valeur *booléenne* true et la *chaîne* "true" sont des choses absolument différentes.

Chapitre 4
Les opérateurs sont sympas

Dans ce chapitre :

- Faisons un peu d'arithmétique.
- Faisons des comparaisons.
- Aller plus loin avec des opérateurs logiques.

Les mathématiciens créent des variables et les manipulent de différentes manières. Ils les additionnent, les multiplient, parfois même les intègrent. Le Chapitre 2 explique comment déclarer et définir des variables, mais il ne dit rien sur la manière de les utiliser afin d'en faire quelque chose. Ce chapitre examine les opérations que l'on peut exécuter avec des variables pour réaliser effectivement quelque chose.



Écrire un programme qui fait vraiment quelque chose, c'est bien. Si vous n'y arrivez pas, vous ne deviendrez jamais un véritable programmeur C#, à moins, bien sûr, que vous ne soyez un consultant, comme moi.

Faire de l'arithmétique

L'ensemble des opérateurs arithmétiques est divisé en plusieurs groupes : les opérateurs arithmétiques simples, les opérateurs d'assignation, et un groupe d'opérateurs spéciaux, propres à la programmation. Une fois que vous les aurez digérés, il vous faudra faire de même pour un autre ensemble d'opérateurs : les opérateurs logiques.

Les opérateurs simples

Les opérateurs simples sont pour la plupart ceux que vous avez appris à l'école primaire. Le Tableau 4.1 en donne la liste :

Tableau 4.1 : Les opérateurs simples.

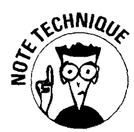
Opérateur	Signification
- (moins unaire)	prendre le négatif de la valeur
*	multiplier
/	diviser
+	additionner
- (moins binaire)	soustraire
%	modulo

La plupart de ces opérateurs sont appelés opérateurs binaires, parce qu'ils opèrent sur deux valeurs : celle qui se trouve du côté gauche de l'opérateur et celle qui se trouve du côté droit. La seule exception est le moins unaire, mais il est aussi simple que les autres :

```
int nl = 5;
int n2 = -nl;  // n2 a maintenant la valeur -5
```

La valeur de -n est le négatif de la valeur de n.

L'opérateur modulo vous est peut-être moins familier que les autres. C'est tout simplement le reste d'une division. Ainsi, 5 % 3 vaut 2, et 25 % 3 vaut 1 (25 - 3 * 8).



La définition stricte de l'opérateur % est : "x = (x / y) + x % y".

Les opérateurs arithmétiques autres que modulo sont définis pour tous les types numériques. L'opérateur modulo n'est pas défini pour les types en virgule flottante, car une division effectuée en virgule flottante n'a pas de reste.

Ordre d'exécution des opérateurs

Il arrive que le sens d'une expression arithmétique ne soit pas parfaitement clair. Par exemple :

```
int n = 5 * 3 + 2;
```

Est-ce que le programmeur veut dire "multiplier 5 par 3 et ajouter 2", ce qui fait 17, ou bien "multiplier 5 par la somme de 3 et 2", ce qui fait 25 ?



C# effectue l'exécution d'une suite d'opérateurs de gauche à droite. Le résultat de l'exemple ci-dessus est donc l'assignation de la valeur 17 à la variable n.

Dans l'exemple suivant, C# détermine la valeur de n en commençant par diviser 24 par 6, puis en divisant le résultat de cette opération par 2 (et non en divisant 24 par le résultat de la division de 6 par 2) :

```
int n = 24 / 6 / 2
```

D'autre part, les opérateurs ont une hiérarchie, ou ordre de priorité. C# commence par examiner l'expression, et exécute les opérations en commençant par celle qui a le niveau de priorité le plus élevé. Dans mes précédents livres, je me suis donné le plus grand mal pour expliquer la priorité des opérateurs, mais je me suis depuis rendu compte que ce n'était qu'une perte de temps (et de neurones). Il vaut toujours mieux se débarrasser de la question de la priorité des opérateurs en utilisant les parenthèses.

De cette façon, la valeur de l'expression suivante est claire, quel que soit l'ordre de priorité des opérateurs :

```
int n = (7 % 3) * (4 + (6 / 3));
```

C# commence par évaluer l'expression qui se trouve dans le bloc de parenthèses le plus profondément enfoui :

```
int n = (7 % 3) * (4 + 2);
```

Cela fait, il remonte vers le bloc de parenthèses le plus englobant, en évaluant chaque bloc l'un après l'autre :

```
int n = 1 * 6;
```

Pour arriver au résultat final :

```
int n = 7
```

Cette règle connaît peut-être une exception. Je trouve ce comportement intolérable, mais de nombreux programmeurs omettent les parenthèses dans des exemples comme le suivant, car tout le monde sait que la priorité de la multiplication est plus élevée que celle de l'addition :

```
int n = 7 + 2 * 3;
```

Le résultat de cette expression et 13 (et non 27).

L'opérateur d'assignation et ses variantes

C# a hérité de C et C++ une idée intéressante : l'assignation y est un opérateur binaire. L'opérateur d'assignation a la valeur de l'argument qui est à sa droite. L'assignation a le même type que les deux arguments qui doivent donc eux-mêmes être de même type.

Cette nouvelle conception de l'opérateur d'assignation n'a aucun effet sur les expressions que vous avez vues jusqu'ici :

```
n = 5 * 3;
```

Dans cet exemple, 5 * 3 vaut 15 et est de type `int`. L'opérateur d'assignation stocke la valeur de type `int` qui se trouve à sa droite dans la variable de type `int` qui se trouve à sa gauche, et retourne la valeur 15. Mais ce n'est pas tout, cette nouvelle conception de l'opérateur d'assignation autorise la forme suivante :

```
m = n = 5 * 3;
```

Les opérateurs d'assignation sont évalués l'un après l'autre, de droite à gauche. Dans cet exemple, le premier à partir de la droite stocke la valeur 15 dans la variable `n`, et retourne 15. Le deuxième et dernier à partir de la droite stocke la valeur 15 dans `m` et retourne 15, qui n'est utilisé par aucun autre opérateur.

Du fait de cette définition étrange de l'opérateur d'assignation, les expressions suivantes, bien qu'étranges, sont licites :

```
int n;
int m;
n = m = 2;
```

C# offre une extension de l'ensemble des opérateurs simples avec un ensemble d'opérateurs construits à partir d'autres opérateurs binaires. Par exemple :

```
n += 1;
```

Cette expression est équivalente à :

```
n = n + 1;
```

Il existe un tel opérateur d'assignation pour pratiquement tous les opérateurs binaires. Je ne sais pas exactement comment ils sont venus au monde, mais pourtant, ils existent.

L'opérateur d'incrémentation

Parmi toutes les additions que l'on peut avoir à faire dans un programme, la plus courante consiste à ajouter 1 à une variable :

```
n = n + 1;
```

Nous avons vu que C# offre le raccourci suivant :

```
n += 1;
```

Mais c'est encore trop compliqué. C# fait encore mieux :

```
++n;  // incrémente n de 1
```

Les trois instructions ci-dessus sont équivalentes. Chacune incrémente `n` de la valeur 1.

L'opérateur d'incrémentation est plutôt bizarre, mais, croyez-le ou non, C# en a en fait deux : ++n, et n++. Le premier, ++n, est l'*opérateur de préincrémentation*, et le second, n++, est l'*opérateur de postincrémentation*. La différence est subtile, mais importante.

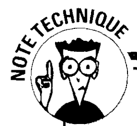
Souvenez-vous que toute expression a un type et une valeur. Dans l'exemple suivant, ++n et n++ sont tous deux de type int :

```
int n;
n = 1;
int o = ++n;
n = 1;
int m = n++;
```

Mais quelles sont les valeurs qui en résultent pour m et o ? (Je vous donne un indice : c'est 1 ou 2.)

La valeur de o est 2, et la valeur de m est 1. Autrement dit, la valeur de l'expression ++n est la valeur de n après avoir été incrémentée, alors que la valeur de l'expression n++ est la valeur de n avant d'avoir été incrémentée. Dans les deux cas, le résultat est 2.

Sur le même principe, il y a des opérateurs de décrémentation (n-- et --n) pour remplacer n = n-1. Ils fonctionnent exactement de la même manière que les opérateurs d'incrémentation.



Pourquoi un opérateur d'incrémentation, et pourquoi en avoir deux ?

L'obscure raison d'être de l'opérateur d'incrémentation vient du fait que le calculateur PDP-8 des années soixante-dix possédait une instruction d'incrémentation. La chose serait aujourd'hui de peu d'intérêt si le langage C, à l'origine de la lignée qui conduit aujourd'hui à C#, n'avait pas été écrit justement pour le PDP-8. Comme cette machine possédait une instruction d'incrémentation, n++ générait moins d'instructions machine que n = n+1. Puisque les machines de l'époque étaient très lentes, on ne ménageait pas ses efforts pour faire l'économie de quelques instructions machine.

Les compilateurs d'aujourd'hui sont plus astucieux, il n'y a davantage de différence entre le temps d'exécution de n++ et celui de n = n+1, donc plus de besoin pour un opérateur

d'incrémentation. Mais les programmeurs sont des créatures qui ont leurs habitudes, et cet opérateur est toujours là. Vous ne verrez presque jamais un programmeur C++ incrémenter une variable en utilisant la forme plus longue mais plus intuitive `n = n+1`. Vous le verrez plutôt utiliser l'opérateur d'incrémentation.

D'autre part, lorsqu'on le rencontre isolé (c'est-à-dire pas à l'intérieur d'une expression plus grande), c'est presque toujours l'opérateur de postincrémentation qui apparaît et non l'opérateur de préincrémentation. Il n'y a aucune raison à cela, en dehors de l'habitude et du fait que ça à l'air plus cool.

Faire des comparaisons – est-ce logique ?

C# comporte également un ensemble d'opérateurs de comparaison logique, montrés par le Tableau 4.2.

Tableau 4.2 : Les opérateurs de comparaison logique.

Opérateur	L'opérateur est vrai si...
a == b	a a la même valeur que b
a > b	a est plus grand que b
a >= b	a est supérieur ou égal à b
a < b	a est plus petit que b
a <= b	a est inférieur ou égal à b
a != b	a n'est pas égal à b

Ces opérateurs sont appelés *opérateurs de comparaison logique*, car ils retournent une valeur de type `bool` `true` ou `false` (vrai ou faux).

Voici un exemple qui fait intervenir une comparaison logique :

```
int m = 5;
int n = 6;
bool b = m > n;
```

Cet exemple assigne la valeur `false` à la variable `b`, car 5 n'est pas plus grand que 6.

Les opérateurs de comparaison logique sont définis pour tous les types numériques, notamment `float`, `double`, `decimal`, et `char`. Tout ce qui suit est licite :

```
bool b;
b = 3 > 2;
b = 3.0 > 2.0;
b = 'a' > 'b';
b = 10M > 12M;
```

Un opérateur de comparaison logique produit toujours un résultat de type `bool`. Il n'est pas valide pour une variable de type `string` (C# offre d'autres moyens de comparer des chaînes).

Comparer des nombres en virgule flottante : qui a le plus gros `float` *?*

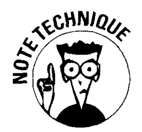
Comparer deux nombres en virgule flottante tient parfois un peu du jeu de hasard, et il faut être très prudent. Considérez les comparaisons suivantes :

```
float f1;
float f2;
f1 = 10;
f2 = f1 / 3;
bool b1 = (3 * f2) == f1;
f1 = 9;
f2 = f1 / 3;
bool b2 = (3 * f2) == f1;
```

La seule différence entre le calcul de `b1` et le calcul de `b2` est la valeur originale de `f1`. Quelles sont donc les valeurs de `b1` et `b2` ? La valeur de b2 est évidemment `true` : 9/3 vaut 3 ; 3 * 3 vaut 9 ; et 9 est égal à 9.

La valeur de `b1` n'est pas aussi évidente : 10/3 vaut 3.333... 3.333... * 3 vaut 9.999... 9.999... est-il égal à 10 ? Ça dépend du niveau intellectuel de votre processeur et de votre compilateur. Avec un Pentium ou un processeur plus récent, C# n'est pas assez malin pour se rendre compte que `b1`

devrait être `true` si le résultat du calcul est un peu décalé par rapport à la comparaison.



Pour faire un peu mieux, vous pouvez utiliser de la façon suivante la fonction de valeur absolue pour comparer `f1` et `f2` :

```
Math.Abs(d1 - 3.0 * d2) < .00001; //choisissez le niveau de précision
```

Cette fonction retourne `true` dans les deux cas. Vous pouvez utiliser la constante `Double.Epsilon` à la place de .00001 pour obtenir le niveau de précision le plus élevé possible. `Epsilon` est la plus petite différence possible entre deux variables de type double qui ne sont pas rigoureusement égales.

Encore plus fort : les opérateurs logiques

Les variables de type `bool` disposent d'un autre ensemble d'opérateurs logiques, définis rien que pour elles, montrés par le Tableau 4.3.

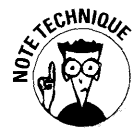
Tableau 4.3 : Les opérateurs logiques.

Opérateur	Retourne `true` si...
!a	a est `false`
a & b	a et b sont `true`
a | b	a ou b ou les deux sont `true` (aussi appelé a et/ou b)
a ^ b	a est `true` ou b est `true` mais pas les deux (aussi appelé a xor b)
a && b	a et b sont `true` avec une évaluation en court-circuit
a || b	a ou b sont `true` avec une évaluation en court-circuit

L'opérateur ! est l'équivalent logique du signe moins. Par exemple, `!a` est `true` si `a` est `false`, et `false` si `a` est `true`.

Les deux opérateurs suivants sont assez clairs. Tout d'abord, `a & b` n'est `true` que si `a` et `b` sont `true`. Et `a | b` est `true` si `a` ou `b` sont `true`. Le signe ^ (aussi appelé le *ou exclusif*) est un peu une bête curieuse. `a^b` est `true` si `a` ou `b` sont `true` mais pas si `a` et `b` sont `true`.

Ces trois opérateurs produisent comme résultat une valeur logique de type `bool`.



Les opérateurs `&`, `|`, et `^` existent aussi dans une version que l'on appelle *opérateur de bits*. Appliqués à une variable de type `int`, ils opèrent bit par bit. Ainsi, 6 & 3 vaut 2 (0110_2 & 0011_2 donne 0010_2), 6 | 3 vaut 7 (0110_2 | 0011_2 donne 0111_2), et 6 ^ 3 vaut 5 (0110_2 ^ 0011_2 donne 0101_2). L'arithmétique binaire est une chose extrêmement sympathique, mais sort du cadre de cet ouvrage.

Les deux derniers opérateurs logiques sont semblables aux trois premiers, mais présentent une différence subtile avec eux. Considérez l'exemple suivant :

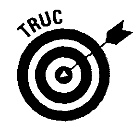
```
bool b = (boolExpression1) & (boolExpression2);
```

Dans ce cas, C# évalue `boolExpression1` et `boolExpression2`, en cherchant à déterminer si l'une et l'autre sont `true` pour en déduire la valeur de `b`. Toutefois, cet effort pourrait être inutile. Si l'une de ces deux expressions est `false`, il n'y a aucune raison d'évaluer l'autre, car quelle qu'en soit la valeur, le résultat de l'ensemble sera `false`.

L'opérateur `&&` permet d'éviter d'évaluer inutilement les deux expressions :

```
bool b = (boolExpression1) && (boolExpression2);
```

Dans ce cas, C# évalue `boolExpression1`. Si elle est `false`, `b` reçoit la valeur `false`, et le programme poursuit son chemin. Si elle est `true`, C# évalue `boolExpression2` et stocke le résultat dans `b`.



L'opérateur `&&` utilise ce que l'on appelle une *évaluation en court-circuit*, car il court-circuite si nécessaire la seconde opération booléenne.

L'opérateur `||` fonctionne sur le même principe :

```
bool b = (boolExpression1) || (boolExpression2);
```

Si `boolExpression1` est `true`, il n'y a aucune raison d'évaluer `boolExpression2`, car le résultat sera `true` de toute façon.

Trouver les âmes sœurs : accorder les types d'expression

Dans un calcul, le type d'une expression est tout aussi important que sa valeur. Examinez l'expression suivante :

```
int n;
n = 5 * 5 + 7;
```

Ma calculatrice me dit que n vaut 32, mais cette expression a aussi un type.

Traduite en termes de types, cette expression devient :

```
int [=] int * int + int;
```

Afin d'évaluer le type d'une expression, suivez le même cheminement que pour en déterminer la valeur. La multiplication a une priorité plus élevée que l'addition. Un int multiplié par un int donne un int. L'addition vient ensuite. Un int plus un int donne un int. On peut donc réduire l'expression ci-dessus de la façon suivante :

```
int * int + int
int + int
int
```

Calculer le type d'une opération

Accorder des types suppose de creuser dans les *sous-expressions*. Chaque expression a un type, et les types du côté gauche et du côté droit d'un opérateur doivent correspondre à ce qui est attendu par celui- ci :

```
type1 <op> type2 @---> type3
```

(La flèche signifie "produit".) type1 et type2 doivent être compatibles avec l'opérateur op.

La plupart des opérateurs admettent différents types. Par exemple, l'opérateur de multiplication :

```
int     * int     @---> int
uint    * uint    @---> uint
long    * long    @---> long
float   * float   @---> float
decimal * decimal @---> decimal
double  * double  @---> double
```

Ainsi, 2 * 3 utilise la version `int * int` de l'opérateur * pour produire 6, de type `int`.

Conversion de type implicite

Tout cela est très bien pour multiplier deux `int` ou deux `float`. Mais qu'arrive-t-il lorsque les deux arguments ne sont pas de même type ? Par exemple, dans ce cas :

```
int n1 = 10;
double d2 = 5.0;
double dResult = n1 * d2;
```

Tout d'abord, C# ne comporte pas d'opération `int * double`. Il pourrait se contenter de produire un message d'erreur, mais il essaie plutôt de comprendre ce qu'a voulu faire le programmeur. C# dispose des versions `int * int` et `double * double` de la multiplication. Il pourrait convertir `d2` en son équivalent `int`, mais il en résulterait la perte de la partie décimale du nombre (ce qui se trouve à droite du point décimal). C# convertit donc en `double` la variable `int n1` et utilise l'opération `double * double`. C'est ce que l'on appelle une *promotion implicite*.

Une promotion implicite est *implicite* parce que C# l'effectue automatiquement, et c'est une *promotion* parce qu'elle transforme une valeur d'un certain type en un type de capacité supérieure. La liste des opérateurs de multiplication donnée à la section précédente apparaît dans l'ordre de promotion croissante de `int` à `double` ou de `int` à `decimal`. Il n'existe aucune conversion implicite entre les types en virgule flottante et le type `decimal`. La conversion d'un type de capacité supérieure tel que double en un type de moindre capacité tel que `int` s'appelle une *rétrogradation*.



Une promotion est aussi appelée *conversion vers le haut*, et une rétrogradation *conversion vers le bas*.



Une rétrogradation (ou conversion vers le bas) implicite n'est pas autorisée. Dans un tel cas, C# génère un message d'erreur.

Conversion de type explicite – le cast

Et si C# se trompait ? Et si le programmeur voulait vraiment effectuer une multiplication en nombres entiers ?

Vous pouvez toujours changer le type d'une variable d'un type valeur en utilisant l'opérateur cast. Un *cast* consiste à mettre entre parenthèses le type désiré et à le placer immédiatement avant la variable ou l'expression concernée.

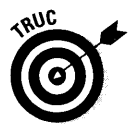
De cette façon, l'expression suivante utilise l'opérateur `int * int` :

```
int n1 = 10;
double d2 = 5.0;
int nResult = n1 * (int)d2;
```

Le cast de `d2` en `int` est une *rétrogradation explicite*, parce que le programmeur a explicitement déclaré son intention.



Vous pouvez faire une conversion explicite entre deux types valeur quels qu'ils soient, que ce soit une promotion ou une rétrogradation.



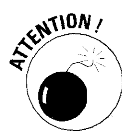
Évitez les conversions de type implicites. Utilisez plutôt un cast avec les types valeur pour faire des conversions explicites.

Laissez la logique tranquille

C# n'offre aucune conversion de type de ou vers le type `bool`.

Assigner un type

Le même principe de compatibilité de types s'applique à l'opérateur d'assignation.



En général, une incompatibilité de type produisant un message d'erreur de compilation se produit dans l'opérateur d'assignation, mais pas à l'endroit qui est la source de l'incompatibilité.

Considérez l'exemple de multiplication suivant :

```
int n1 = 10;
int n2 = 5.0 * n1;
```

La deuxième ligne de cet exemple génère un message d'erreur dû à une incompatibilité de type, mais l'erreur se produit lors de l'assignation, et non lors de la multiplication. En voici la terrible histoire : afin d'effectuer la multiplication, C# convertit implicitement `n1` en `double`. C# peut alors effectuer une multiplication en type `double`, dont le résultat est dans le tout-puissant type `double`.

Toutefois, les types de ce qui est à droite et à gauche de l'opérateur d'assignation doivent être compatibles, mais le type de ce qui est à gauche ne peut pas changer, car C# n'accepte pas de rétrograder implicitement une expression. Le compilateur génère donc le message d'erreur suivant : `Impossible de convertir implicitement le type double en int.`

C# autorise cette expression avec un cast explicite :

```
int n1 = 10;
int n2 = (int)(5.0 * n1);
```

(Les parenthèses sont nécessaires parce que l'opérateur de cast a un niveau de priorité très élevé.) Cela fonctionne. La variable `n1` est promue en double, la multiplication est effectuée, et le résultat en double est rétrogradé en `int`. Toutefois, on peut alors se demander si le programmeur est sain d'esprit, car il aurait été beaucoup plus facile pour lui comme pour le compilateur d'écrire 5 * `n1`.

L'opérateur ternaire, le redoutable

La plupart des opérateurs admettent deux arguments, certains n'en admettent qu'un, et un seul en admet trois : l'opérateur ternaire. Celui-ci est redoutable et pour une bonne raison. Il a le format suivant :

```
bool expression ? expression1 : expression2
```

Et je rendrai les choses encore plus confuses avec un exemple :

```
int a = 1;
int b = 2;
int nMax = (a > b) ? a : b;
```

Dans cet exemple, si a est plus grand que b, la valeur de l'expression est a. Si a n'est pas plus grand que b, la valeur de l'expression est b.

L'opérateur ternaire est impopulaire pour plusieurs raisons. Tout d'abord, il n'est pas nécessaire. Utiliser le type d'une instruction if (que nous décrirons au Chapitre 5) a le même effet et est plus facile à comprendre. D'autre part, l'opérateur ternaire donne une véritable expression, quel que soit son degré de ressemblance avec un type ou un autre d'instruction if. Par exemple, les expressions 1 et 2 doivent être de même type. Il en résulte ceci :

```
int a = 1;
double b = 0.0;
int nMax = (a > b) ? a : b;
```

Cette instruction ne se compile pas, alors que nMax aurait dû recevoir la valeur de a. Comme a et b doivent être de même type, a est promu en double pour être compatible avec b. Le type qui résulte de ?: est maintenant double, qui doit être explicitement rétrogradé en int pour que l'assignation soit possible :

```
int a = 1;
double b = 0.0;
int nMax;
//ceci fonctionne
nMax = (int)((a > b) ? a : b);
//de même que ceci
nMax = (a > b) ? a : (int)b;
```

Vous aurez rarement l'occasion de voir une utilisation de l'opérateur ternaire.

Chapitre 5

Contrôler le flux d'exécution d'un programme

Dans ce chapitre :

Prendre une décision si vous le pouvez.

Décider quoi faire d'autre.

Faire des boucles sans tourner en rond.

Utiliser la boucle `while`.

Utiliser la boucle `for`.

Considérez le très simple programme suivant :

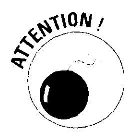
```
using System;
namespace HelloWorld
{
  public class Class1
  {
    //le programme commence ici
    static void Main(string[] args)
    {
      //demande son nom à l'utilisateur
      Console.WriteLine("Entrez votre nom:");
      //lit le nom entré par l'utilisateur
      string sName = Console.ReadLine();
      //accueille l'utilisateur par son nom
      Console.WriteLine("Hello, " + sName);
      //attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
```

```
        }
    }
}
```

En dehors du fait qu'il présente quelques aspects fondamentaux de la programmation en C#, ce programme n'a pratiquement aucun intérêt. Il ne fait qu'afficher ce que vous avez entré. On peut imaginer un exemple un peu plus compliqué qui prendrait les données saisies par l'utilisateur, ferait quelques calculs avec elles et afficherait un résultat (sinon pourquoi faire des calculs ?), puis se terminerait. Toutefois, même un tel programme ne peut avoir qu'une utilité limitée.

L'une des caractéristiques les plus importantes de n'importe quel processeur est sa capacité de prendre des décisions. Par "prendre des décisions", je veux dire que le processeur oriente l'exécution du programme vers un chemin d'instructions si une certaine condition est vraie, et vers un autre chemin dans le cas contraire. Tout langage de programmation doit comporter cette capacité fondamentale pour contrôler le flux d'exécution des programmes.

Il y a trois types de base d'instructions de contrôle de flux : l'instruction `if`, la boucle, et le saut.



L'une des instructions de boucle, `foreach`, est décrite au Chapitre 6.

Contrôler le flux d'exécution

La base de la capacité de prise de décision de C# est l'instruction `if` :

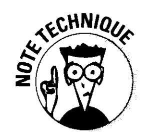
```
if (bool expression)
{
    // l'exécution est orientée ici si l'expression est vraie
}
// l'exécution se poursuit ici, que l'expression soit vraie ou non
```

Les parenthèses qui suivent immédiatement l'instruction `if` contiennent une instruction de type `bool` (pour en savoir plus sur les expressions de type `bool`, reportez-vous au Chapitre 4). Juste après cette expression, il y a un bloc de code, délimité par une paire de parenthèses. Si l'expression est vraie, le programme exécute ce bloc de code. Si l'expression n'est pas vraie, il ignore ce bloc de code et passe directement à ce qui suit.

L'instruction if est plus facile à comprendre avec un exemple concret :

```
// garantir que a n'est pas inférieur à 0
// si a est inférieur à 0. . .
if (a < 0)
{
    // . . . alors, assigner 0 à a
    a = 0;
}
```

Ce fragment de code permet de garantir que la variable a est toujours supérieure ou égale à zéro. L'instruction if dit : "Si a est inférieur à 0, alors assigner 0 à a."



Les parenthèses sont facultatives. C# traite "if (expression booléenne) instruction" exactement de la même manière que "if (expression booléenne) {instruction}". Le consensus général (auquel je souscris) est de toujours utiliser les parenthèses. Autrement dit, faites-le.

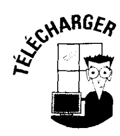
Et si j'ai besoin d'un exemple ?

Imaginez un petit programme qui calcule des intérêts. L'utilisateur entre le principal et le taux d'intérêt, et le programme donne la valeur qui en résulte à la fin de l'année (ce n'est pas un programme très sophistiqué). En C#, ce calcul tout simple apparaît comme suit :

```
//calcul de la valeur du principal
//plus l'intérêt
decimal mInterestPaid;
mInterestPaid = mPrincipal * (mInterest / 100);
//calcule maintenant le total
decimal mTotal = mPrincipal + mInterestPaid;
```

La première équation multiplie le principal, mPrincipal, par le taux d'intérêt, mInterest (divisé par 100, car le taux d'intérêt est généralement exprimé en pourcentage), pour obtenir l'intérêt à payer, mInterestPaid. L'intérêt à payer est alors ajouté au principal, ce qui donne le nouveau principal, stocké dans la variable mTotal.

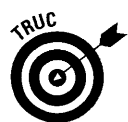
Le programme doit être capable de répondre à presque tout ce qu'un être humain est capable d'entrer. Par exemple, on ne peut pas accepter un principal ou un intérêt négatif. Le programme CalculateInterest ci-dessus contient des vérifications pour éviter ce genre de choses :




```
// CalculateInterest -
//             calcule le montant de l'intérêt
//             à payer pour un principal donné. Si le
//             principal ou le taux d'intérêt est négatif,
//             produit un message d'erreur.
using System;
namespace CalculateInterest
{
  public class Class1
  {
    public static int Main(string[] args)
    {
      //demande à l'utilisateur d'entrer le principal initial
      Console.Write("Entrez le principal :");
      string sPrincipal = Console.ReadLine();
      decimal mPrincipal = Convert.ToDecimal(sPrincipal);
      //vérifie que le principal n'est pas négatif
      if (mPrincipal < 0)
      {
        Console.WriteLine("Le principal ne peut pas être négatif");
        mPrincipal = 0;
      }
      //demande à l'utilisateur d'entrer le taux d'intérêt
      Console.Write("Entrez le taux d'intérêt :");
      string sInterest = Console.ReadLine();
      decimal mInterest = Convert.ToDecimal(sInterest);
      //vérifie que le taux d'intérêt n'est pas négatif.
      if (mInterest < 0)
      {
        Console.WriteLine("Le taux d'intérêt doit être positif");
        mInterest = 0;
      }
      //calcule la valeur du principal
      //plus l'intérêt
      decimal mInterestPaid;
      mInterestPaid = mPrincipal * (mInterest / 100);
      //calcule maintenant le total
      decimal mTotal = mPrincipal + mInterestPaid;
      //affiche résultat
      Console.WriteLine(); // skip a line
      Console.WriteLine("Principal      = "  + mPrincipal);
      Console.WriteLine("Taux d'intérêt = "  + mInterest + "%");
      Console.WriteLine();
      Console.WriteLine("Interêt payé   = "  + mInterestPaid);
      Console.WriteLine("Total          = "  + mTotal);
      //attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
```

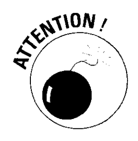
```
            return 0;
          }
        }
      }
```

Le programme CalculateInterest commence par demander son nom à l'utilisateur en utilisant l'instruction WriteLine() pour écrire une chaîne sur la console.



Dites à l'utilisateur exactement ce que vous voulez. Si possible, indiquez aussi le format que vous voulez. Les utilisateurs donnent rarement de bonnes réponses à une invite aussi peu claire que >.

Notre exemple utilise la commande ReadLine() pour lire sous forme d'une chaîne de caractères ce que tape l'utilisateur au clavier jusqu'à la touche Entrée. Comme ce programme attend le principal dans le type decimal, la chaîne entrée doit être convertie avec la commande Convert.ToDecimal(). Le résultat est alors stocké dans mPrincipal.



Les commandes ReadLine(), WriteLine(), et ToDecimal(), sont toutes des exemples d'*appels de fonction*. Je décrirai en détail les appels de fonctions au Chapitre 6, mais ceux-là sont assez immédiatement compréhensibles. Vous devriez avoir au moins une idée de ce dont il s'agit. Si mes lumineuses explications ne sont pas assez lumineuses pour vous, vous pouvez les ignorer et aller voir le Chapitre 6.

La ligne suivante effectue la vérification de mPrincipal. Si sa valeur est négative, le programme annonce sans ménagement à l'utilisateur qu'il a fait une ânerie. Il fait ensuite la même chose pour le taux d'intérêt. Cela fait, il effectue le calcul de l'intérêt, très simple, que nous avons déjà vu plus haut, et affiche le résultat en utilisant une série de commandes WriteLine().

Le programme affiche les résultats suivants, sur la base d'un principal légitime et d'un taux d'intérêt usuraire, curieusement légal en bien des contrées :

```
Entrez le principal :1234
Entrez le taux d'intérêt :21

Principal      = 1234
Taux d'intérêt = 21%

Interêt payé   = 259.14
```

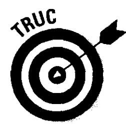
```
Total          = 1493.14
Appuyez sur Entrée pour terminer...
```

Avec une entrée invalide, le programme produit la réponse suivante :

```
Entrez le principal :1234
Entrez le taux d'intérêt :-12.5
Le taux d'intérêt doit être positif

Principal      = 1234
Taux d'intérêt = 0%

Interêt payé   = 0
Total          = 1234
Appuyez sur Entrée pour terminer...
```



Pour que le source soit plus lisible, mettez en retrait les lignes d'une instruction `if`. C# ne tient pas compte de la mise en retrait. Beaucoup d'éditeurs de code comporte une fonction de mise en retrait automatique : chaque fois que vous tapez la commande `if`, le texte correspondant est mis en retrait automatiquement. Pour activer cette fonction dans Visual Studio, sélectionnez Outils/Options, et cliquez sur le dossier Éditeur de texte. Parmi les sous-dossiers de celui-ci, sélectionnez C#, et, dans ce dernier, Tabulations. Dans cette page, sélectionnez Mise en retrait Intelligente, et dans la zone Tabulations, spécifiez la taille du retrait que vous voulez (en nombre d'espaces). Pour ce livre, j'ai utilisé une mise en retrait de deux espaces.

Qu'est-ce que je peux faire d'autre ?

Certaines fonctions ont besoin de tester des conditions mutuellement exclusives. Par exemple, le segment de code suivant stocke le plus élevé de deux nombres, `a` et `b` dans la variable `max` :

```
// stocke dans max le plus élevé de a et b
int max;
// si a est plus grand que b. . .
if (a > b)
{
    //  . . . conserve a comme maximum
    max = a;
}
// si a est inférieur ou égal à b. . .
```

```
if (a <= b)
{
    // . . . conserve b comme maximum
    max = b;
}
```

La seconde instruction `if` est ici inutile, car les deux conditions sont mutuellement exclusives. Si `a` est plus grand que `b`, alors il ne peut pas être inférieur ou égal à `b`. C'est pour les situations de ce type qu'il y a dans C# une instruction `else`.

Le mot-clé `else` définit un bloc de code qui sera exécuté si l'expression logique contenue dans l'instruction `if` n'est pas vraie.

Notre calcul de maximum devient maintenant :

```
// stocke dans max le plus élevé de a et b
int max;
// si a est plus grand que b. . .
if (a > b)
{
    // . . .conserve a comme maximum; sinon . . .
    max = a;
}
else
{
    // . . . conserve b comme maximum
    max = b;
}
```

Si `a` est plus grand que `b`, c'est le premier bloc de code qui est exécuté. Dans le cas contraire, c'est le second. Au bout du compte, `max` contient la valeur du plus grand de `a` ou `b`.

Éviter même le `else`

Les séquences de plusieurs clauses `else` peuvent donner une certaine confusion. Certains programmeurs, dont moi-même, préfèrent les éviter lorsque ça permet de faire un code plus clair. On pourrait écrire le calcul du maximum de la façon suivante :

```
// stocke dans max le plus élevé de a et b
int max;
```

```
// suppose que a est plus grand que b
max = a;
// si ce n'est pas le cas. . .
if (b > a)
{
  // ...alors, on peut changer d'avis
  max = b;
}
```

Il y a des programmeurs qui évitent ce style comme la peste, et je peux les comprendre, mais ça ne veut pas dire que je vais faire comme eux. Je me contente de les comprendre. Les deux styles, avec ou sans "`else`", sont couramment utilisés, et vous les rencontrerez souvent.

Instructions `if` *imbriquées*

Le programme `CalculateInterest` prévient l'utilisateur en cas d'entrée invalide, mais il ne semble pas très pertinent de poursuivre le calcul de l'intérêt si l'une des valeurs est invalide. Ça ne peut guère tirer à conséquence ici, parce que le calcul de l'intérêt est pratiquement immédiat et parce que l'utilisateur peut en ignorer le résultat, mais il y a bien des calculs qui sont loin d'être aussi rapides. De plus, pourquoi demander à l'utilisateur un taux d'intérêt s'il a déjà entré une valeur invalide pour le principal ? Il sait bien que le résultat du calcul sera invalide, quelle que soit la valeur qu'il saisit maintenant.

Le programme ne devrait donc demander le taux d'intérêt à l'utilisateur que si la valeur du principal est valide, et n'effectuer le calcul de l'intérêt que si les deux valeurs sont valides. Pour réaliser cela, il vous faut deux instructions `if`, l'une dans l'autre.



Une instruction `if` placée dans le corps d'une autre instruction `if` est appelée une instruction *imbriquée*.

Le programme suivant, `CalculateInterestWithEmbeddedTest`, utilise des instructions `if` imbriquées pour éviter les questions inutiles si un problème est détecté avec les valeurs entrées.




```
// CalculateInterestWithEmbeddedTest -
//              calcule le montant de l'intérêt à
//              payer pour un principal donné. Si
//              le principal ou le taux d'intérêt est
//              négatif, alors génère un message d'erreur
```

```
//              et n'effectue pas le calcul.
using System;
namespace CalculateInterestWithEmbeddedTest
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      //définit un taux d'intérêt maximum
      int nMaximumInterest = 50;
      //demande à l'utilisateur d'entrer le principal initial
      Console.Write("Entrez le principal :");
      string sPrincipal = Console.ReadLine();
      decimal mPrincipal = Convert.ToDecimal(sPrincipal);
      //si le principal est négatif. . .
      if (mPrincipal < 0)
      {
        // . . .génère un message d'erreur. . .
        Console.WriteLine("Le principal ne peut pas être négatif");
      }
      else
      {
        //  . . .sinon, demande le taux d'intérêt
        Console.Write("Entrez le taux d'intérêt :");
        string sInterest = Console.ReadLine();
        decimal mInterest = Convert.ToDecimal(sInterest);
        //si le taux d'intérêt est négatif ou trop élevé. . .
        if (mInterest < 0 || mInterest > nMaximumInterest)
        {
          //  . . .génère un autre message d'erreur
          Console.WriteLine("Le taux d'intérêt doit être positif " +
                            "et pas supérieur à " + nMaximumInterest);
          mInterest = 0;
        }
        else
        {
          //le principal et l'intérêt sont tous deux valides
          //calcule donc la valeur du principal
          //plus l'intérêt
          decimal mInterestPaid;
          mInterestPaid = mPrincipal * (mInterest / 100);
          //calcule maintenant le total
          decimal mTotal = mPrincipal + mInterestPaid;
          //affiche résultat
          Console.WriteLine();  // skip a line
          Console.WriteLine("Principal     = " + mPrincipal);
          Console.WriteLine("Taux d'intérêt = " + mInterest + "%");
          Console.WriteLine();
```

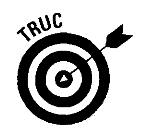
```
            Console.WriteLine("Intérêt payé   = " + mInterestPaid);
            Console.WriteLine("Total          = " + mTotal);
          }
        }
        //attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
      }
    }
  }
```

Le programme commence par lire la valeur du principal entrée par l'utilisateur. Si elle est négative, il affiche un message d'erreur et se termine. Dans le cas contraire, le contrôle passe à la clause `else`, et le programme poursuit son exécution.

Dans cet exemple, la vérification du taux d'intérêt a été améliorée. Le programme demande ici un taux d'intérêt qui ne soit pas négatif (règle mathématique) et qui soit inférieur à un maximum (règle juridique). Cette instruction `if` utilise un test composé :

```
if (mInterest < 0 || mInterest > nMaximumInterest)
```

Cette expression est vraie si `mInterest` est inférieur à zéro ou si `mInterest` est plus grand que `nMaximumInterest`. Remarquez que j'ai déclaré `nMaximumInterest` en haut du programme au lieu de le *coder localement* sous forme de constante.



Définissez toujours au début de votre programme les constantes importantes.

Placer les constantes dans des variables au début du programme est utile à plusieurs titres. Tout d'abord, chaque constante a ainsi un nom : `nMaximumInterest` est beaucoup plus descriptif que 50. D'autre part, elles sont beaucoup plus faciles à retrouver dans les expressions. Enfin, il est beaucoup plus aisé d'en changer la valeur en cas de nécessité. Remarquez que c'est `nMaximumInterest` qui apparaît dans le message d'erreur. Si vous remplacez `nMaximumInterest` par 60, par exemple, cette modification n'affecte pas seulement le test, mais aussi le message d'erreur.

Si l'utilisateur entre une valeur correcte pour le principal mais un taux d'intérêt négatif, le programme affiche :

```
Entrez le principal :1234
Entrez le taux d'intérêt :-12.5
```

```
Le taux d'intérêt doit être positif et pas supérieur à 50.
Appuyez sur Entrée pour terminer...
```

Ce n'est que si l'utilisateur entre des valeurs correctes pour le principal et pour le taux d'intérêt que le programme effectue le calcul demandé :

```
Entrez le principal :1234
Entrez le taux d'intérêt :12.5

Principal      = 1234
Taux d'intérêt = 12.5%

Interêt payé   = 154.25
Total          = 1388.25
Appuyez sur Entrée pour terminer...
```

Les commandes de boucle

L'instruction `if` permet à un programme de s'orienter sur un chemin ou sur un autre dans le code en cours d'exécution, selon la valeur d'une expression `bool`éenne. Elle permet de faire des programmes incomparablement plus intéressants que ceux qui sont dépourvus de capacité de décision. Ajoutez maintenant la capacité d'exécuter un ensemble d'instructions de façon itérative, et vous aurez fait un autre saut qualitatif dans la capacité de vos programmes.

Considérez le programme `CalculateInterest` que nous avons vu plus haut dans ce chapitre. On pourrait faire la même chose avec une calculatrice ou même à la main avec un crayon et un papier, en se donnant moins de mal que pour écrire et exécuter un programme.

Et si vous pouviez calculer le montant du principal pour chaque période d'un an successive ? Ce serait beaucoup plus utile. Une simple macro de feuille de calcul Excel serait toujours plus facile à réaliser, mais au moins, il y a un progrès.

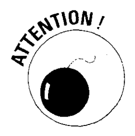
Ce qu'il vous faut, c'est un moyen pour l'ordinateur d'exécuter plusieurs fois la même séquence d'instructions. C'est ce qu'on appelle une *boucle*.

Commençons par la boucle de base, `while`

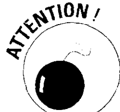
Le mot-clé de C# `while` permet d'exécuter une boucle de la forme la plus simple :

```
while(bool expression)
{
    //l'exécution est répétée tant que l'expression reste vraie
}
```

La première fois que l'instruction de boucle `while` est rencontrée, l'expression booléenne est évaluée. Si elle est vraie, le code contenu dans le bloc qui suit est exécuté. Lorsque l'accolade qui en indique la fin est rencontrée, l'exécution reprend à l'instruction `while`. Dès que l'expression booléenne est fausse, le bloc de code qui suit est ignoré, et l'exécution du programme passe directement à ce qui suit.

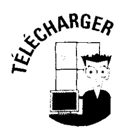


Si la condition n'est pas vraie la première fois que l'instruction `while` est rencontrée, le bloc de code qui suit n'est jamais exécuté.



Les programmeurs s'expriment souvent de façon un peu bizarre (il sont d'ailleurs bizarres la plupart du temps). Un programmeur pourrait dire qu'une boucle est exécutée jusqu'à ce qu'une certaine condition soit fausse. Pour moi, cela voudrait dire que le contrôle passe en dehors de la boucle dès que la condition devient fausse, quel que soit le point où il en est de son exécution à ce moment-là. Ce n'est évidemment pas comme ça que ça se passe. Le programme ne vérifie si la condition est vraie ou non que lorsque le contrôle de l'exécution arrive effectivement en haut de la boucle.

Vous pouvez utiliser l'instruction `while` pour réaliser le programme `CalculateInterestTable`, qui est une version en boucle du programme `CalculateInterest`. `CalculateInterestTable` calcule une table des valeurs du principal pour chaque année, en mettant en évidence l'accumulation des intérêts annuels :




```
// CalculateInterestTable - calcul de l'intérêt cumulé
//             payé sur la base d'un principe déterminé
//             sur une période de plusieurs années
using System;
namespace CalculateInterestTable
{
  using System;
```

```
public class Class1
  {
  public static void Main(string[] args)
  {
    //demande à l'utilisateur d'entrer le principal initial
    Console.Write("Entrez le principal :");
    string sPrincipal = Console.ReadLine();
    decimal mPrincipal = Convert.ToDecimal(sPrincipal);
    //si le principal est négatif. . .
    if (mPrincipal < 0)
    {
      // . . .génère un message d'erreur. . .
      Console.WriteLine("Le principal ne peut pas être négatif");
    }
    else
    {
      //  . . .sinon, demande le taux d'intérêt
      Console.Write("Entrez le taux d'intérêt :");
      string sInterest = Console.ReadLine();
      decimal mInterest = Convert.ToDecimal(sInterest);
      //si le taux d'intérêt est négatif.. . .
      if (mInterest < 0)
      {
        //  . . .génère un autre message d'erreur
        Console.WriteLine("Le taux d'intérêt doit être positif");
        mInterest = 0;
      }
      else
      {
        //le principal et le taux d'intérêt sont valides
        //demande donc le nombre d'années
        Console.Write("Entrez le nombre d'années :");
        string sDuration = Console.ReadLine();
        int nDuration = Convert.ToInt32(sDuration);
        //vérifie la valeur entrée
        Console.WriteLine();  // écrit une ligne blanche
        Console.WriteLine("Principal      = " + mPrincipal);
        Console.WriteLine("Taux d'intérêt = " + mInterest + "%");
        Console.WriteLine("Durée          = " + nDuration + "ans");
        Console.WriteLine();
        //effectue une boucle selon le nombre d'années spécifié
        int nYear = 1;
        while(nYear <= nDuration)
        {
          //calcule la valeur du principal
          //plus l'intérêt
          decimal mInterestPaid;
          mInterestPaid = mPrincipal * (mInterest / 100);
```

```
                //calcule maintenant le nouveau principal en ajoutant
                //l'intérêt au principal précédent
                mPrincipal = mPrincipal + mInterestPaid;
                //arrondit le principal au centime le plus proche
                mPrincipal = decimal.Round(mPrincipal, 2);
                //affiche le résultat
                Console.WriteLine(nYear + "-" + mPrincipal);
                //passe à l'année suivante
                nYear = nYear + 1;
              }
            }
          }
          //attend confirmation de l'utilisateur
          Console.WriteLine("Appuyez sur Entrée pour terminer...");
          Console.Read();
        }
      }
    }
```

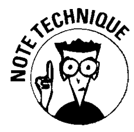
L'essai d'exécution de ce programme donne ce qui suit :

```
Entrez le principal :1234
Entrez le taux d'intérêt :12.5
Entrez le nombre d'années :10

Principal      = 1234
Taux d'intérêt = 12.5%
Durée          = 10 ans

1-1388.25
2-1561.78
3-1757
4-1976.62
5-2223.7
6-2501.66
7-2814.37
8-3166.17
9-3561.94
10-4007.18
Appuyez sur Entrée pour terminer...
```

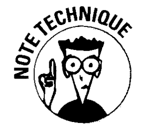
Chaque valeur représente le principal total à l'issue du nombre d'années écoulées, sur la base d'un cumul annuel d'intérêt simple. Par exemple, un principal initial de 1 234 € à 12,5 % donne 3 561,94 € au bout de neuf ans.



La plupart des valeurs comportent deux décimales pour les centimes. Comme les zéros de la partie décimale ne sont pas affichés, certaines valeurs n'ont qu'un chiffre ou même aucun après la virgule. Ainsi, 12,70 est affiché comme 12,7. Vous pouvez y remédier en utilisant les caractères de mise en forme décrits au Chapitre 9.

Le programme `CalculateInterestTable` commence par lire la valeur du principal et celle du taux d'intérêt entrées par l'utilisateur, et par vérifier quelles sont valides. Il lit ensuite le nombre d'années sur lequel effectuer l'itération, et stocke cette valeur dans la variable `nDuration`.

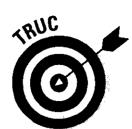
Avant d'entrer dans la boucle `while`, le programme déclare une variable `nYear`, qu'il initialise à la valeur 1. Ce sera "l'année en cours", c'est-à-dire que cette valeur va changer "chaque année" à chaque boucle successive exécutée par le programme. Si le numéro de l'année contenu dans `nYear` est inférieur à la durée totale contenue dans `nDuration`, le principal pour "l'année en cours" est recalculé en utilisant l'intérêt calculé sur la base de "l'année précédente". Le principal calculé est affiché avec le numéro de l'année correspondante.



L'instruction `decimal.Round()` arrondit la valeur calculée au centime le plus proche.

La clé du fonctionnement de ce programme se trouve dans la dernière ligne du bloc. L'instruction `nYear = nYear + 1;` incrémente `nYear` de 1. Si `nYear` a la valeur 3 avant cette instruction, elle aura la valeur 4 après. Cette incrémentation fait passer le calcul d'une année à la suivante.

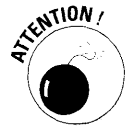
Une fois que l'année a été incrémentée, le contrôle revient en haut de la boucle, où la valeur de `nYear` est comparée à la durée demandée. Dans l'exemple exécuté ci-dessus, si le numéro de l'année en cours est inférieur ou égal à 10, le calcul continue. Après avoir été incrémentée dix fois, la valeur de `nYear` devient 11, qui est plus grand que 10, et le contrôle du programme passe à l'instruction qui suit immédiatement la boucle `while`. Autrement dit, il sort de la boucle.



La plupart des commandes de boucle suivent ce même principe de base qui consiste à incrémenter une variable servant de compteur jusqu'à ce qu'elle dépasse une valeur fixée.

La variable `nYear` servant de compteur dans `CalculateInterestTable` doit être déclarée et initialisée avant la boucle `while` dans laquelle elle est utilisée. En outre, l'incrémentation de la variable `nYear` doit généralement être la dernière instruction de la boucle. Comme le montre cet exemple,

vous devez prévoir de quelles variables vous aurez besoin. Ce procédé vous sera plus facile à manier une fois que vous aurez écrit quelques milliers de boucles `while`, comme moi.



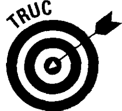
Lorsque vous écrivez une boucle `while`, n'oubliez pas d'incrémenter la variable servant de compteur, comme je l'ai fait dans cet exemple :

```
int nYear = 1;
while (nYear < 10)
{
    // . . .instructions. . .
}
```

J'ai omis l'instruction `nYear = nYear + 1;`. Sans l'incrémentation, la valeur de `nYear` est toujours 1, et le programme continue à exécuter la boucle sans jamais s'arrêter. C'est ce qu'on appelle une boucle infinie. La seule manière d'en sortir est d'arrêter le programme (ou de redémarrer l'ordinateur).



Faites attention à ce que la condition de sortie de la boucle puisse réellement être satisfaite. En général, il suffit pour cela que la variable compteur soit correctement incrémentée. Sans cette précaution, vous êtes bon pour la boucle infinie et l'utilisateur rancunier.



Comme une boucle infinie est une faute assez courante, ne soyez pas trop vexé si vous vous y laissez prendre.

Et maintenant, `do... while`

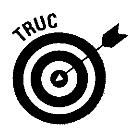
Il existe une variante de `while` : c'est la boucle `do... while`. Avec elle, la condition n'est évaluée qu'à la fin de la boucle :

```
int nYear = 1;
do
{
    // . . .instructions. . .
    nYear = nYear + 1;
} while (nYear < nDuration);
```

À la différence de la boucle `while`, la boucle `do... while` est exécutée au moins une fois, quelle que soit la valeur de `nDuration`. Toutefois, ce type de boucle est assez peu utilisé en pratique.

Briser une boucle, c'est facile

Il existe deux instructions de contrôle spéciales que vous pouvez utiliser dans une boucle : `break` et `continue`. La commande `break` fait passer le contrôle à la première expression qui suit la boucle dans laquelle elle se trouve. La commande `continue` fait passer le contrôle directement à l'expression conditionnelle en haut de la boucle afin de recommencer de la manière appropriée.

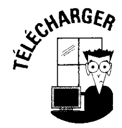


J'ai rarement utilisé `continue` dans ma carrière de programmeur, et je doute qu'il y ait beaucoup de programmeurs qui se souviennent seulement de son existence. Ne l'oubliez tout de même pas complètement. Elle vous servira peut-être un jour pour jouer au Scrabble.

Par exemple, imaginez que vous vouliez récupérer votre argent à la banque dès que le principal dépasse un certain nombre de fois le montant initial, indépendamment du nombre d'années écoulées. Vous pouvez facilement résoudre ce problème en ajoutant ce qui suit dans la boucle :

```
if (mPrincipal > (maxPower * mOriginalPrincipal))
{
  break;
}
```

La commande `break` ne sera exécutée que lorsque la condition de l'instruction `if` sera vraie. Dans ce cas, lorsque la valeur calculée du principal sera supérieure à `maxPower` multiplié par la valeur initiale du principal. L'exécution de la commande `break` fait passer le contrôle en dehors de la boucle `while(nYear <= nDuration)`, et le programme poursuit son exécution jusqu'à sa fin.



Vous trouverez sur le site Web une version du calcul de table d'intérêt qui comporte cette adjonction (il serait un peu long d'en donner le source ici).

Voici un exemple de résultats affichés par ce programme :

```
Entrez le principal :100
Entrez le taux d'intérêt :25
Entrez le nombre d'années :100

Principal       = 100
Taux d'intérêt = 25%
Durée           = 100 ans
```

```
Arrêter si la valeur initiale est multipliée par 10

1-125
2-156.25
3-195.31
4-244.14
5-305.18
6-381.48
7-476.85
8-596.06
9-745.08
10-931.35
11-1164.19
Appuyez sur Entrée pour terminer...
```

Le programme se termine dès que le principal calculé dépasse 1 000 ¤. Comme vous voyez, il est plus performant que la Belle au bois dormant.

Faire des boucles jusqu'à ce qu'on y arrive

Le programme `CalculateInterestTable` est assez malin pour se terminer si l'utilisateur entre une valeur invalide, mais c'est tout de même un peu dur pour l'utilisateur de le planter là sans autre forme de procès. Même mon peu sympathique programme de comptabilité me donne droit à trois essais pour entrer mon mot de passe avant de me laisser tomber.



Une combinaison de `while` et `break` permet de donner au programme un peu plus de souplesse. Le programme `CalculateInterestTableMoreForgiving` en montre le principe :

```
// CalculateInterestTableMoreForgiving - calcule l'intérêt
//          payé sur un nombre d'années déterminé. Cette
//          version donne à l'utilisateur 3 possibilités
//          un principal et un taux d'intérêt valides.
using System;
namespace CalculateInterestTableMoreForgiving
{
  using System;
  public class Class1
  {
    public static void Main(string[] args)
    {
      // définit un taux d'intérêt maximal
      int nMaximumInterest = 50;
```

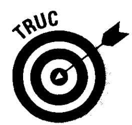
```
// demande à l'utilisateur le principal initial; continue
// jusqu'à ce qu'une valeur valide soit entrée
decimal mPrincipal;
while(true)
{
  Console.Write("Entrez le principal :");
  string sPrincipal = Console.ReadLine();
  mPrincipal = Convert.ToDecimal(sPrincipal);
  // sort de la boucle si valeur entrée est valide
  if (mPrincipal >= 0)
  {
    break;
  }
  // génère un message d'erreur si valeur entrée est invalide
  Console.WriteLine("Le principal ne peut pas être négatif");
  Console.WriteLine("Veuillez recommencer");
  Console.WriteLine();
}
// demande maintenant à l'utilisateur le taux d'intérêt
decimal mInterest;
while(true)
{
  Console.Write("Entrez le taux d'intérêt :");
  string sInterest = Console.ReadLine();
  mInterest = Convert.ToDecimal(sInterest);
  // n'accepte pas un taux d'intérêt négatif ou trop grand...
  if (mInterest >= 0 && mInterest <= nMaximumInterest)
  {
    break;
  }
  //  . . .et génère aussi un message d'erreur
  Console.WriteLine("Le taux d'intérêt doit être positif " +
                  "et pas supérieur à " + nMaximumInterest);
  Console.WriteLine("Veuillez recommencer");
  Console.WriteLine();
}
// l'intérêt comme le principal sont valides,
// demande donc le nombre d'années
Console.Write("Entrez le nombre d'années :");
string sDuration = Console.ReadLine();
int nDuration = Convert.ToInt32(sDuration);
// vérifie la valeur entrée
Console.WriteLine();  // écrit une ligne blanche
Console.WriteLine("Principal      = " + mPrincipal);
Console.WriteLine("Taux d'intérêt = " + mInterest + "%");
Console.WriteLine("Durée          = " + nDuration + " ans");
Console.WriteLine();
// effectue une boucle sur le nombre d'années spécifié
```

```
      int nYear = 1;
      while(nYear <= nDuration)
      {
        // calcule la valeur du principal
        // plus l'intérêt
        decimal mInterestPaid;
        mInterestPaid = mPrincipal * (mInterest / 100);
        // calcule maintenant le nouveau principal en ajoutant
        // l'intérêt au précédent principal
        mPrincipal = mPrincipal + mInterestPaid;
        // arrondit le principal au centime le plus proche
        mPrincipal = decimal.Round(mPrincipal, 2);
        // affiche le résultat
        Console.WriteLine(nYear + "-" + mPrincipal);
        // passe à l'année suivante
        nYear = nYear + 1;
      }
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

Ce programme fonctionne largement de la même manière que les exemples précédents, sauf pour ce qui est entré par l'utilisateur. Dans ce cas, c'est une boucle `while` qui remplace les instructions `if` utilisées précédemment pour détecter les entrées invalides. Par exemple :

```
decimal mPrincipal;
while(true)
{
  Console.Write("Entrez le principal :");
  string sPrincipal = Console.ReadLine();
  mPrincipal = Convert.ToDecimal(sPrincipal);
  // sort de la boucle si valeur entrée est valide
  if (mPrincipal >= 0)
  {
    break;
  }
  // génère un message d'erreur si valeur entrée est invalide
  Console.WriteLine("Le principal ne peut pas être négatif");
  Console.WriteLine("Veuillez recommencer");
  Console.WriteLine();
}
```

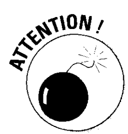
Cette portion de code reçoit une valeur de l'utilisateur à l'intérieur d'une boucle. Si la valeur entrée est satisfaisante, le programme sort de la boucle et poursuit son exécution. Si la valeur est incorrecte, un message d'erreur est envoyé à l'utilisateur, et le contrôle repasse au début de la boucle de saisie.



Vous pouvez le voir de cette façon : "Le programme ne sort pas de la boucle tant que l'utilisateur n'a pas répondu correctement."

Remarquez que la condition a été inversée, car il ne s'agit plus qu'une réponse incorrecte produise un message d'erreur, mais qu'une réponse correcte fasse sortir de la boucle. Dans la partie concernant la saisie du taux d'intérêt, par exemple, le test `Principal < 0 || mPrincipal > nMaximumInterest` devient `mInterest >= 0 && mInterest <= nMaximumInterest`. Il est clair que `mInterest >= 0` est le contraire de `mInterest < 0`. Ce qui n'est peut-être pas aussi évident est que le `OR ||` est remplacé par un `AND &&`. Autrement dit : "Sortir de la boucle si le taux d'intérêt est supérieur à zéro *et* inférieur au montant maximum."

Dernier point à noter : la variable `mPrincipal` doit être déclarée en dehors de la boucle, pour des questions de règles sur la portée des variables, que j'expliquerai dans la section suivante de ce chapitre.



Vous allez peut-être trouver cela évident, mais l'expression `true` est évaluée comme `true`. Par conséquent, `while (true)` est l'archétype de la boucle infinie. C'est la commande `break` qu'elle contient qui fait sortir de la boucle. Aussi, si vous utilisez une boucle `while (true)`, faites particulièrement attention à ce que la condition de `break` puisse être satisfaite.

Voici un exemple de résultat d'exécution de ce programme :

```
Entrez le principal :-1000
Le principal ne peut pas être négatif
Veuillez recommencer

Entrez le principal :1000
Entrez le taux d'intérêt :-10
Le taux d'intérêt doit être positif et pas supérieur à 50
Veuillez recommencer

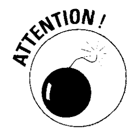
Entrez le taux d'intérêt :10
Entrez le nombre d'années :5

Principal      = 1000
```

```
Taux d'intérêt = 10%
Durée          = 5 ans

1-1100
2-1210
3-1331
4-1464.1
5-1610.51
Appuyez sur Entrée pour terminer...
```

Le programme n'accepte ni principal négatif ni taux d'intérêt négatif, et m'explique patiemment mon erreur chaque fois.



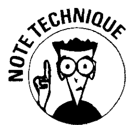
Expliquez toujours exactement son erreur à l'utilisateur avant de lui demander à nouveau d'entrer une valeur.

Les règles de portée des variables

Une variable déclarée dans le corps d'une boucle n'est définie que dans cette boucle. Examinez ce fragment de code :

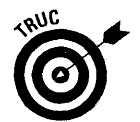
```
int nDays = 1;
while(nDays < nDuration)
{
    int nAverage = nValue / nDays;
    //  . . .instructions . . .
    nDays = nDays + 1;
}
```

La variable `nAverage` n'est pas définie en dehors de la boucle `while`. Il y a différentes raisons à cela, mais considérez celle-ci : lors de la première exécution de la boucle, le programme rencontre la déclaration `int nAverage`, et la variable est définie. Lors de la seconde exécution de la boucle, le programme rencontre à nouveau la déclaration de `nAverage`. S'il n'y avait pas les règles de portée des variables, ce serait une erreur, car la variable est déjà définie.



Il y a d'autres raisons, plus convaincantes que celle-ci, mais je m'en tiendrai là pour le moment.

Il me suffit de dire que la variable `nAverage` disparaît, aux yeux de C#, dès que le programme atteint l'accolade fermante qui indique la fin de la boucle.



Un programmeur expérimenté dit que la portée de la variable nAverage est limitée à la boucle while.

Comprendre la boucle la plus utilisée : for

La boucle while est la plus simple des structures de boucle de C#, et la plus utilisée après for.

Une boucle for a la structure suivante :

```
for(initExpression; condition; incrementExpression)
{
    // . . .instructions. . .
}
```

Lorsqu'une boucle for est rencontrée, le programme commence par exécuter initExpression, puis il évalue la condition. Si la condition est vraie, le programme exécute les instructions qui constituent le corps de la boucle, lequel est délimité par les accolades qui suivent immédiatement l'instruction for. Lorsque l'accolade fermante est atteinte, le contrôle passe à l'exécution de incrementExpression, puis à nouveau à l'évaluation de la condition, et la boucle recommence aussi longtemps que la condition de for reste vraie.

En fait, la définition d'une boucle for peut être convertie dans la boucle while suivante :

```
initExpression;
while(condition)
{
    // . . .instructions. . .
    incrementExpression;
}
```

Un exemple de boucle for

Un exemple vous permettra de mieux comprendre le fonctionnement d'une boucle for :

```
// instructions C#
a = 1;
```

```
// et maintenant une boucle
for(int nYear = 1; nYear < nDuration; nYear = nYear + 1)
{
    //  . . .corps de la boucle . . .
}
// le programme continue ici
a = 2;
```

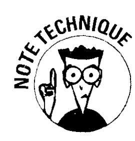
Supposez que le programme vienne d'exécuter l'instruction `a = 1;`. Il déclare ensuite la variable `nYear` et l'initialise à 1. Cela fait, il compare `nYear` à `nDuration`. Si `nYear` est plus petit que `nDuration`, le corps de la boucle (les instructions contenues dans les accolades) est exécuté. Lorsqu'il rencontre l'accolade fermante, le programme revient en haut de la boucle, et exécute l'expression `nYear = nYear + 1` avant d'effectuer la comparaison `nYear < nDuration`.

Pourquoi auriez-vous besoin d'une autre boucle ?

À quoi peut bien servir une boucle `for` si C# permet de faire la même chose avec une boucle `while` ? La réponse la plus simple est qu'elle ne sert à rien. Une boucle `for` n'ajoute rien à ce qu'une boucle `while` permet déjà de faire.

Toutefois, les différentes parties de la boucle `for` existent par commodité, et pour différencier clairement les trois parties que toute boucle doit comporter : l'initialisation, le critère de sortie, et l'incrémentation. Non seulement c'est plus facile à lire, mais c'est aussi plus difficile à rater (souvenez-vous que les erreurs les plus courantes dans une boucle `while` sont d'oublier d'incrémenter la variable compteur et de ne pas définir correctement le critère de sortie).

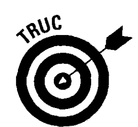
Indépendamment de tout alibi justificateur, la raison la plus importante de comprendre la boucle `for` est que c'est celle que tout le monde utilise, donc celle que vous allez voir neuf fois sur dix quand vous lirez du code écrit par quelqu'un d'autre.



La boucle `for` est conçue de telle sorte que la première expression initialise une variable compteur, et la dernière l'incrémente. Toutefois, le langage C# n'impose pas cette règle. Vous pouvez faire ce que vous voulez dans ces deux parties de l'instruction, mais sachez que vous seriez mal inspiré d'y faire autre chose.

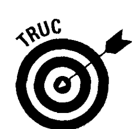
L'opérateur d'incrémentation est particulièrement populaire dans les boucles `for` (je décris l'opérateur d'incrémentation, ainsi que d'autres, au Chapitre 4). Une boucle `for` pour notre exemple de calcul des intérêts cumulés pourra s'écrire ainsi :

```
for(int nYear = 1; nYear < nDuration; nYear++)
{
    //  . . .corps de la boucle . . .
}
```



C'est presque toujours l'opérateur de postincrémentation que vous verrez dans une boucle `for`, plutôt que l'opérateur de préincrémentation, bien que l'effet en soit le même dans ce cas. Il n'y a pas d'autres raisons à cela que l'habitude et le fait que ça a l'air plus cool. (On m'a dit que ça marchait très bien pour briser la glace. Rien n'est moins sûr, mais vous pouvez toujours essayer d'exiber votre code, à tout hasard.)

La boucle `for` a aussi une variante dont je ne peux pas faire semblant de comprendre la raison d'être. Si la condition logique est omise, elle est considérée comme vraie. Par conséquent, `for(;;)` produit une boucle infinie.



Vous verrez effectivement `for(;;)` utilisée pour réaliser une boucle infinie beaucoup plus souvent que `while(true)`. Pourquoi ? Je n'en ai pas la moindre idée.

Des boucles imbriquées

Une boucle peut être placée à l'intérieur d'une autre boucle :

```
for( . . .condition . . .)
{
  for( . . .autre condition . . .)
  {
    //  . . .corps de la boucle . . .
  }
}
```

Une boucle incluse dans une autre est entièrement exécutée à chaque passage de la boucle qui la contient.



Une boucle incluse dans une autre boucle est appelée une boucle *imbriquée*.

Des boucles imbriquées ne peuvent pas être "entrelacées". Par exemple, ce qui suit n'est pas possible :

```
do              //début d'une boucle do
{
  for( . . .)     //début d'une boucle for
  {
  } while( . . .) //fin de la boucle do.. while
}                 //fin de la boucle for
```

Je ne suis même pas très sûr de ce que ça voudrait dire, mais c'est sans importance, puisque de toute façon c'est illicite.

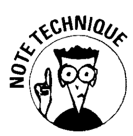


Une instruction `break` dans une boucle imbriquée ne fait sortir que de la boucle dans laquelle elle se trouve.

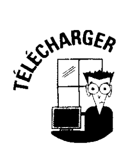
Dans l'exemple suivant, l'instruction `break` fait sortir de la boucle B, et revenir à la boucle A :

```
// boucle for A
for( . . .condition . . .)
{
  // boucle for B
  for( . . .autre condition . . .)
  {
    //  . . .corps du code de la boucle . . .
    if (. . .condition . . .)
    {
      break;        //fait sortir de la boucle B mais pas de A
    }
  }
}
```

C# n'a pas de commande `break` qui fasse sortir simultanément des deux boucles.



Ce n'est pas une limitation aussi importante qu'il y paraît. En pratique, la logique souvent complexe de telles boucles imbriquées est mieux encapsulée dans une fonction. L'exécution d'un `return` à l'intérieur de n'importe quelle boucle fait alors sortir de la fonction, donc de toutes les boucles imbriquées, quelle que soit la profondeur à laquelle on peut se trouver. Je décrirai les fonctions au Chapitre 7.



Le saugrenu programme DisplayXWithNestedLoops utilise deux boucles imbriquées pour afficher un grand X sur la console de l'application :

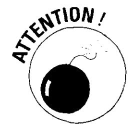
```
// DisplayXWithNestedLoops - utilise deux boucles imbriquées
//                           pour dessiner un X
using System;
namespace DisplayXWithNestedLoops
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      int nConsoleWidth = 40;
      // itère sur les lignes du "cadre"
      for(int nRowNum = 0;
          nRowNum < nConsoleWidth;
          nRowNum += 2)
      {
        // itère maintenant sur les colonnes
        for (int nColumnNum = 0;
             nColumnNum < nConsoleWidth;
             nColumnNum++)
        {
          // le caractère par défaut est un espace
          char c = ' ';
          // si le numéro de la ligne est égal à celui de la colonne...
          if (nColumnNum == nRowNum)
          {
            // . . .remplace l'espace par un backslash
            c = '\\';
          }
          // si la colonne est du côté opposé de la ligne...
          int nMirrorColumn = nConsoleWidth - nRowNum;
          if (nColumnNum == nMirrorColumn)
          {
            // . . .remplace l'espace par un slash
            c = '/';
          }
          // affiche le caractère correspondant à l'intersection
          // de la ligne et de la colonne
          Console.Write(c);
        }
        Console.WriteLine();
      }
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
```

```
            }
        }
    }
```

Ce programme commence par définir un nombre arbitraire de lignes et de colonnes, représentant la taille du X à dessiner. Si vous augmentez ce nombre, le X sort de la fenêtre de l'application.

Ce programme utilise une boucle `for` pour réaliser l'itération sur les lignes du X. À l'intérieur de celle-ci, il entre dans une seconde boucle `for` qui réalise l'itération sur les colonnes de l'affichage. Ce procédé définit une matrice d'affichage. Il ne reste plus qu'à décider quelles cellules de la matrice recevront un espace, ce qui les rendra invisibles, et lesquelles recevront un caractère. Remplissez les bonnes cellules, et vous aurez un X.

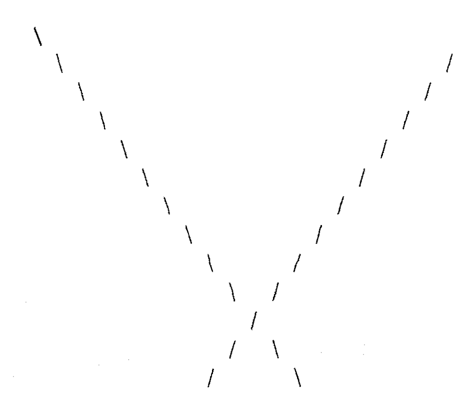
Le programme commence par définir une variable `c` de type `char`, qu'il initialise avec un espace qui sera sa valeur par défaut. Il compare ensuite le numéro de la ligne et celui de la colonne. S'ils sont égaux, il remplace l'espace par une barre oblique inverse (backslash).

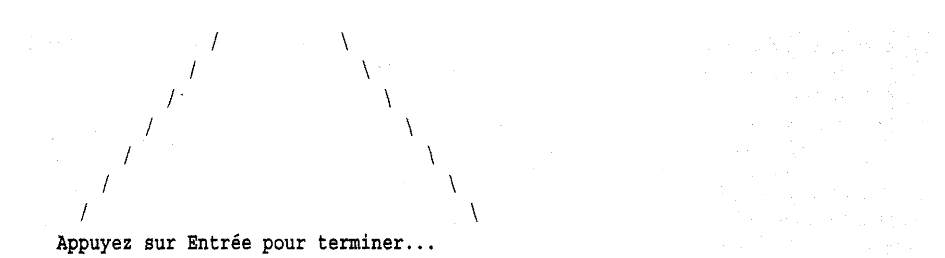


Souvenez-vous que le backslash est utilisé pour indiquer les caractères spéciaux. Par exemple, '\n' est le caractère de nouvelle ligne. Le caractère spécial '\\' est le backslash.

Par lui-même, le remplacement de l'espace lorsque le numéro de la ligne est égal au numéro de la colonne trace une ligne du coin supérieur gauche de la matrice au coin inférieur droit. Pour obtenir un effet miroir, le programme place une barre oblique ('/') lorsque le numéro de la colonne symétrique est égal au numéro de la ligne.

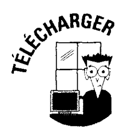
Ce qui donne le résultat suivant :





```
          /        \
         /          \
        /            \
       /              \
      /                \
     /                  \
    /                    \
Appuyez sur Entrée pour terminer...
```

Il y a des choses plus utiles, mais c'est amusant.



Si vous voulez être sérieux, allez voir l'exemple `DisplaySin`, qui utilise le même genre de logique pour afficher verticalement une ligne sinusoïdale dans la fenêtre de l'application. Je suis peut-être un excité (et même certainement), mais j'aime beaucoup ce programme. C'est sur des programmes de ce genre qu'il m'est arrivé de me casser les dents.

L'instruction de contrôle `switch`

Il vous arrivera souvent de vouloir tester la valeur d'une variable. Par exemple, `nMaritalStatus` pourrait valoir 0 pour signifier "célibataire", 1 pour "marié", 2 pour "divorcé", 3 pour "veuf", et 4 pour "c'est pas vos oignons". Afin de reconnaître ces différents cas, vous pouvez utiliser une série d'instructions `if` :

```
if (nMaritalStatus == 0)
{
  //doit être célibataire
  //  . . .instructions. . .
}
else
{
  if (nMaritalStatus == 1)
  {
    //doit être marié
    //  . . .autres instructions . . .
```

Et ainsi de suite.

Vous pouvez vous rendre compte que la répétition de ces instructions `if` est un peu fastidieuse. Il est si courant d'avoir à tester des cas multiples que C# offre une structure spéciale pour faire un choix dans un ensemble

de conditions mutuellement exclusives. Cette instruction s'appelle `switch` et fonctionne de la façon suivante :

```
switch(nMaritalStatus)
{
  case 0:
          //  . . . instructions si célibataire. . .
          break;
  case 1:
          //  . . . instructions si marié. . .
          break;
  case 2:
          //  . . . instructions si divorcé. . .
          break;
  case 3:
          //  . . . instructions si veuf. . .
          break;
  case 4:
          //  . . . allez vous rhabiller. . .
          break;
  default:
          //passe ici quand aucun cas ne correspond ;
          //c'est probablement une condition d'erreur
          break;
  break;
}
```

L'expression qui se trouve en haut de l'instruction `switch` est évaluée. Dans ce cas, c'est simplement la variable `nMaritalStatus`. La valeur de cette expression est alors comparée à celle qui suit chaque mot-clé `case`. Si elle ne correspond à aucun de ces cas, le contrôle passe directement à la condition `default`.

L'argument de l'instruction `switch` peut aussi être de type `string` :

```
string s = "Davis";
switch(s)
{
  case "Davis":
          //  . . . le contrôle passera par ici. . .
          break;
  case "Smith":
          //  . . . instructions si marié. . .
          break;
  case "Jones":
          //  . . . instructions si divorcé. . .
```

```
            break;
    case "Hvidsten":
            // . . . instructions si veuf. . .
            break;
    default:
            // passe ici quand aucun cas ne correspond
            break;
}
```



L'utilisation de l'instruction `switch` comporte quelques contraintes sévères :

- L'argument de `switch()` doit être d'un type admis comme compteur ou de type `string`.
- Les valeurs en virgule flottante sont exclues.
- Les valeurs de `case` doivent être de même type que l'expression de `switch`.
- Les valeurs de `case` doivent être des constantes au sens où leur valeur doit être connue lors de la compilation (une instruction telle que `case x` est illicite, à moins que `x` ne soit une constante).
- Chaque clause `case` doit se terminer par une instruction `break` (ou autre commande de sortie dont nous n'avons pas encore parlé, comme `return`). Cette commande de sortie fait sortir le contrôle de l'instruction `switch`.

Cette règle a toutefois une exception : une même clause `case` peut comporter plusieurs fois le mot-clé `case`, comme dans l'exemple suivant :

```
string s = "Davis";
switch(s)
{
  case "Davis":
  case "Hvidsten":
        //fait la même chose pour Davis ou Hvidsten
        //car leur situation est la même
          break;
  case "Smith":
          // . . . instructions si marié. . .
          break;
  default:
          // passe ici quand aucun cas ne correspond
          break;
}
```

Ce procédé permet au programme d'exécuter les mêmes opérations, que le contenu de la chaîne soit "Davis" ou "Hvidsten".

Le modeste goto

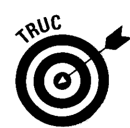
Vous pouvez aussi transférer le contrôle d'une manière non structurée en utilisant l'instruction goto. Elle est suivie par l'un des éléments suivants :

- Une étiquette.
- Un case d'une instruction switch.
- Le mot-clé default, représentant la clause par défaut d'une instruction switch.

Le fragment de code suivant montre comment est utilisée l'instruction goto :

```
//si la condition est vraie. . .
if (a > b)
{
  //  . . .le contrôle passe de goto à l'étiquette spécifiée
  goto exitLabel;
}
//  . . .quel que soit le code qui se trouve ici. . .
exitLabel:
  //le contrôle passe ici
```

L'instruction goto est impopulaire, pour les mêmes raisons qui en font une commande de contrôle si puissante : elle est presque entièrement dépourvue de structure. Si vous l'utilisez, il peut être extrêmement difficile de maîtriser le flux de l'exécution au-delà d'un petit morceau de code particulièrement trivial.



L'utilisation de goto a déclenché quasiment des guerres de religion. En fait, le langage C# lui-même a été critiqué pour avoir adopté cette instruction. En réalité, goto n'est ni si horrible ni nécessaire. Comme vous pourrez presque toujours éviter de vous en servir, je vous recommande de vous en tenir à bonne distance.

Troisième partie

Programmation et objets



Dans cette partie...

Une chose est de déclarer une variable ici et là pour faire des additions et des soustractions ; tout autre chose est d'écrire de véritables programmes que les gens peuvent utiliser (des gens ordinaires, mais des gens). Dans cette partie, vous allez découvrir comment regrouper des données et faire des opérations sur ces données. Ce sont les connaissances de base nécessaires à tout travail de programmation, que vous verrez souvent dans les offres d'emploi.

Chapitre 6

Rassembler des données : classes et tableaux

Dans ce chapitre :

Les classes en C#.

Stocker des données dans un objet.

Assigner et utiliser une référence à un objet.

Créer et générer un tableau d'objets.

Vous pouvez librement déclarer et utiliser tous les types intrinsèques, tels que `int`, `double` et `bool`, afin de stocker les informations nécessaires à vos programmes. Pour certains programmes, de si simples variables ne suffisent pas. Toutefois, nombre de programmes ont besoin de rassembler sous forme d'ensembles pratiques les données qui sont en relation les unes avec les autres.

Certains programmes ont besoin de rassembler les données qui appartiennent logiquement à un même groupe mais ne sont pas pour autant de même type. Par exemple, une application utilisée par une université traite des étudiants, chacun ayant son nom, la moyenne de ses notes, et son numéro d'identification. Logiquement, le nom peut être de type `string`, la moyenne des notes de type `double`, et le numéro d'identification de type `long`. Un programme de ce type a besoin de réunir toutes ces variables de types différents dans une même structure nommée `Student`. Heureusement, C# offre une structure appelée *classe* qui permet de regrouper des variables de types différents.

Dans d'autres cas, un programme aura besoin de rassembler une série d'objets de même type. Prenez par exemple un programme qui calcule la moyenne générale des notes d'un étudiant sur l'ensemble d'un cycle d'études. Comme on veut que la précision du résultat final ne soit pas affectée par l'arrondi des moyennes intermédiaires, le type `double` est ce qui convient le mieux pour la moyenne de chaque matière pour chaque année. Il faudra donc une forme ou une autre de collection de variables de type `double` afin de contenir toutes les moyennes annuelles pour chaque matière. C'est dans ce but que C# permet de réaliser un *tableau*.

Enfin, un véritable programme de traitement des données sur les étudiants aura besoin de définir des groupes d'étudiants par diplôme. Un tel programme devra alors faire fusionner la notion de classe et la notion de tableau pour réaliser un tableau d'étudiants. Par la magie de la programmation en C#, c'est ce que vous pouvez faire aussi.

Montrez votre classe

Une classe est une réunion de données et de fonctions dissemblables, dans un même petit ensemble bien ordonné. C# vous donne la liberté de faire des classes aussi mal fichues que vous voulez, mais une classe a pour but de représenter un *concept*.

Les analystes disent : "Une classe introduit dans le programme une carte du problème à résoudre." Par exemple, imaginez que vous vouliez réaliser un simulateur de trafic. Celui-ci va représenter le trafic, dans le but de réaliser de nouvelles rues, avec des intersections ou même des autoroutes. J'aimerais bien que vous fassiez un simulateur de trafic qui résoudrait le problème de l'intersection devant chez moi.

Toute description d'un problème concernant le trafic comporterait le terme *véhicule*. Un véhicule a une vitesse maximale, qui doit avoir sa place dans les équations. Il a aussi un poids, et certains sont purement et simplement des épaves. D'autre part, un véhicule peut démarrer et s'arrêter. La notion de *véhicule* fait donc partie du problème à résoudre.

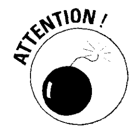
Un bon programme de simulation de trafic en C# comprendrait nécessairement la classe `Vehicle`, dans laquelle seraient décrites les propriétés significatives d'un véhicule. La classe `Vehicle` aurait des propriétés telles que `dTopSpeed`, `nWeight`, et `bClunker`. Je parlerai des propriétés `stop` et `go` au Chapitre 7.

Définir une classe

La classe Vehicle pourrait par exemple se présenter ainsi :

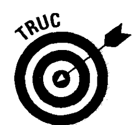
```
public class Vehicle
{
  public string sModel;         // nom du modèle
  public string sManufacturer;  // nom du constructeur
  public int nNumOfDoors;       // nombre de portes du véhicule
  public int nNumOfWheels;      // nombre de roues du véhicule
}
```

La définition d'une classe commence par les mots public class, suivis du nom de la classe, dans ce cas, Vehicle.



C# fait la différence entre les majuscules et les minuscules dans les noms de classe, comme pour tous les autres noms utilisés en C#. C# n'impose aucune règle sur les noms de classe, mais il existe une règle non officielle selon laquelle le nom d'une classe doit commencer par une majuscule.

Le nom d'une classe est suivi par une accolade ouvrante et une accolade fermante. Entre ces deux accolades apparaissent les *membres* que comporte éventuellement cette classe. Les membres d'une classe sont des variables qui en constituent les éléments. Dans cet exemple, la classe Vehicle commence par le membre string sModel, qui contient le modèle du véhicule. Si c'est une voiture particulière, le nom du modèle pourrait être "Eldorado". Vous en voyez certainement tous les jours. Le second membre de notre exemple de classe Vehicle est string sManufacturer, qui contient naturellement le nom du constructeur. Enfin, les deux dernières propriétés sont le nombre de portes et le nombre de roues du véhicule.



Comme pour toute variable, donnez aux membres d'une classe des noms aussi descriptifs que possible. Dans l'exemple ci-dessus, j'ai ajouté des commentaires à la déclaration de chaque membre, mais ce n'était pas nécessaire. Le nom de chaque variable dit clairement de quoi il s'agit.

L'attribut public qui précède le nom de la classe rend celle-ci universellement accessible en tout endroit du programme. De même, l'attribut public placé devant le nom d'un membre de la classe le rend tout aussi accessible en tout endroit du programme. On peut également utiliser d'autres attributs. Le Chapitre 11 traite en détail la question de l'accessibilité.

Une définition de classe doit décrire les propriétés d'un objet qui joue un rôle incontournable dans le problème à résoudre. C'est un peu difficile à faire dans l'immédiat, parce que vous ne savez pas encore quel est le problème, mais je suppose que vous voyez où je veux en venir.

Quel est notre objet ?

Définir une classe `Vehicle` n'est pas la même chose que de construire une voiture. Vous n'aurez pas ici à emboutir de la tôle ni à visser des écrous. Un objet classe se déclare de façon semblable, mais pas tout à fait identique, à un objet intrinsèque.



D'une façon générale, le terme objet signifie "quelque chose". Ça ne nous aide pas beaucoup. Une variable `int` est un objet `int`. Un véhicule est un objet `Vehicle`. Vous-même, vous êtes un objet lecteur. Quant à moi, je suis un auteur...

Le fragment de code suivant crée une voiture de la classe `Vehicle` :

```
Vehicle myCar;
myCar = new Vehicle();
```

La première ligne déclare une variable `myCar` de type `Vehicle`, tout comme vous auriez pu déclarer un objet `nQuelqueChose` de la classe `int`. La commande `new Vehicle ( )` crée un objet de type `Vehicle` et le stocke dans la variable `myCar`. Le `new` n'a rien à voir avec l'âge de `myCar`. Cette commande crée une nouvelle zone de mémoire dans laquelle votre programme pourra stocker les propriétés de `myCar`.

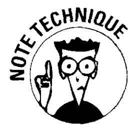


Dans la terminologie C#, on dira que `myCar` est un objet de la classe `Vehicle`, mais aussi que `myCar` est une instance de `Vehicle`. Dans ce contexte, *instance* signifie "un exemple de", ou "un exemplaire de". On peut aussi utiliser ce terme sous forme de verbe, en parlant d'*instancier* un `Vehicle`.

Comparez la déclaration de `myCar` avec celle de la variable entière `num` :

```
int num;
num = 1;
```

La première ligne déclare la variable num, et la deuxième ligne stocke dans l'emplacement défini par la variable num une constante déjà existante de type int.

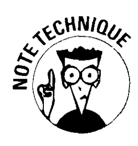


Il y a en fait une différence dans la manière de stocker en mémoire l'objet intrinsèque num et l'objet myCar. La constante 1 n'occupe pas de mémoire, car le CPU et le compilateur C# savent déjà l'un et l'autre ce qu'est un "1". Mais votre CPU ne sait pas ce qu'est un Vehicle. L'expression new Vehicle alloue l'espace mémoire nécessaire à la description d'un objet Vehicle pour le CPU, pour C#, et pour le reste du monde.

Accéder aux membres d'un objet

Tout objet de la classe Vehicle a ses propres membres. L'expression suivante stocke le nombre 1 dans le membre nNumberOfDoors de l'objet référencé par myCar :

```
myCar.nNumberOfDoors = 1;
```



Toute opération en C# doit être évaluée par type aussi bien que par valeur. L'objet myCar est un objet du type Vehicle. La variable Vehicle.nNumberOfDoors est de type int (voyez la définition de la classe Vehicle). Comme la constante 5 est aussi de type int, le type de ce qui est à droite de l'opérateur d'assignation est le même que celui de ce qui est à gauche.

De même, le code suivant stocke une référence aux chaînes décrivant le modèle et le nom du constructeur de myCar :

```
myCar.sManufacturer = "BMW";        // ne perdez pas espoir
myCar.sModel = "Izeta";             // c'est une époque disparue
```

(L'Izeta était une petite voiture construite pendant les années cinquante, dont l'unique porte constituait toute la face avant.)

Soyons ringards : pourquoi s' embêter avec des classes ?

Avec le temps, l'édifice des classes a pris de l'importance dans les langages de programmation. Si vous examinez la chaîne que forment les principaux langages, et leurs périodes de popularité maximale, vous pouvez y distinguer le schéma suivant :

- Fortran (de la fin des années cinquante au début des années quatre-vingt) : pas de notion de classe.
- C (de la fin des années soixante-dix au début des années quatre-vingt dix) : les classes ne sont utilisées qu'à des fins d'organisation. Il est possible d'écrire des programmes qui n'en font aucun usage.
- C++ (du milieu des années quatre-vingt à aujourd'hui) : la notion de classe y est beaucoup plus évoluée. Il est toujours possible d'écrire des programmes qui ne s'en servent pas, mais seulement en se limitant à un sous-ensemble du langage.
- Java (du milieu des années quatre-vingt-dix à aujourd'hui) : la notion de classe y est fondamentale. Impossible d'écrire du code sans y avoir recours.
- C# (aujourd'hui) : comme Java.

La notion de classe a pris une importance croissante parce que les programmeurs se sont rendu compte que les classes étaient très efficaces pour représenter des objet du monde réel. Imaginez par exemple que je sois en train d'écrire un programme de gestion de comptes bancaires. Un compte bancaire présente des caractéristiques telles que le nom de son titulaire, le numéro du compte, le solde, et le nom de la banque. Or, je sais bien que ces propriétés font partie d'un même ensemble, car elles décrivent toutes le même objet : un compte de ma banque. Connaître le solde sans connaître le numéro du compte correspondant, par exemple, n'a aucun sens.

En C#, je peux créer une classe `BankAccount`, associée à une variable `string` contenant le nom du titulaire, une variable `int` contenant le numéro du compte, une variable `double` ou `decimal`, contenant le solde, une variable `string` contenant le nom de la banque, et ainsi de suite. Un seul objet de la classe `BankAccount` contient donc toutes les propriétés, pertinentes pour mon problème, d'un compte bancaire donné.

Un exemple de programmes à base d'objets

Le très simple programme suivant, `VehicleDataOnly` :

- Définit la classe `Vehicle`.
- Crée un objet `myCar`.
- Assigne les propriétés de `myCar`.
- Récupère ces valeurs dans l'objet pour les afficher.




```
// VehicleDataOnly - crée un objet de type Vehicle,
//                   donne une valeur à ses membres à partir des
//                   saisies de l'utilisateur, et affiche le tout
using System;
namespace VehicleDataOnly
{
  public class Vehicle
  {
    public string sModel;        // nom du modèle
    public string sManufacturer; // nom du constructeur
    public int nNumOfDoors;     // nombre de portes du véhicule
    public int nNumOfWheels;    // nombre de roues du véhicule
  }
  public class Class1
  {
    // C'est ici que commence le programme
    static void Main(string[] args)
    {
      // demande son nom à l'utilisateur
      Console.WriteLine("Entrez les propriétés de votre véhicule");
      // crée une instance de Vehicle
      Vehicle myCar = new Vehicle();
      // utilise une variable pour donner une valeur à un membre
      Console.Write("Nom du modèle = ");
      string s = Console.ReadLine();
      myCar.sModel = s;
      // on peut aussi donner une valeur à un membre directement
      Console.Write("Nom du constructeur = ");
      myCar.sManufacturer = Console.ReadLine();
      // lecture du reste des données
      Console.Write("Nombre de portes = ");
      s = Console.ReadLine();
      myCar.nNumOfDoors = Convert.ToInt32(s);
      Console.Write("Nombre de roues = ");
      s = Console.ReadLine();
```

```
            myCar.nNumOfWheels = Convert.ToInt32(s);
            // affiche maintenant les résultats
            Console.WriteLine("\nVotre véhicule est une ");
            Console.WriteLine(myCar.sManufacturer + " " + myCar.sModel);
            Console.WriteLine("avec " + myCar.nNumOfDoors + " portes, "
                        + "sur " + myCar.nNumOfWheels
                        + " roues");
            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
        }
    }
}
```

Le source de ce programme commence par une définition de la classe `Vehicle`.



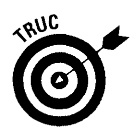
La définition d'une classe peut être placée avant ou après `Class1`. C'est sans importance. Adoptez simplement un style, et gardez-le.

Le programme crée un objet `myCar` de la classe `Vehicle`, puis remplit chacune de ces propriétés en lisant ce qui est saisi au clavier par l'utilisateur. Il n'y a pas de vérification de la validité des données. Le programme restitue alors à l'écran, dans un format légèrement différent, les données saisies par l'utilisateur.

L'exécution de ce programme affiche les résultats de la façon suivante :

```
Entrez les propriétés de votre véhicule
Nom du modèle = Metropolitan
Nom du constructeur = Nash
Nombre de portes = 2
Nombre de roues = 4

Votre véhicule est une
Nash Metropolitan
avec 2 portes, sur 4 roues
Appuyez sur Entrée pour terminer...
```



À la différence de `ReadLine()`, les appels à `Read()` laissent le curseur à la fin de la chaîne affichée. La saisie de l'utilisateur apparaît donc sur la même ligne que l'invite. D'autre part, l'ajout du caractère de nouvelle ligne, '\n', produit une ligne blanche à l'affichage, sans avoir à utiliser pour cela `WriteLine()`.

Distinguer les objets les uns des autres

Un constructeur automobile sait identifier sans erreur chaque voiture qu'il produit. De même, un programme peut créer de nombreux objets de la même classe :

```
Vehicle car1 = new Vehicle();
car1.sManufacturer = "Studebaker";
car1.sModel = "Avanti";
// ce qui suit est sans effet sur car1
Vehicle car2 = new Vehicle();
car2.sManufacturer = "Hudson";
car2.nVehicleNamePart = "Hornet";
```

Si vous créez un objet `car2` et que vous lui assignez le nom de constructeur "Hudson", ça n'aura aucun effet sur la Studebaker `car1`.

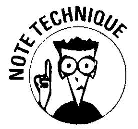
La capacité de distinguer les objets les uns des autres constitue une partie de la puissance de la notion de classe. L'objet associé à la 2 CV Citroën peut être créé, manipulé ou même ignoré, comme une entité à part entière, distincte des autres objets, y compris l'Avanti (bien que ce soient toutes deux des classiques).

Pouvez-vous me donner des références ?

L'opérateur point et l'opérateur d'assignation sont les deux seuls opérateurs définis sur les types de référence :

```
// crée une référence nulle
Vehicle yourCar;
// assigne une valeur à la référence
yourCar = new Vehicle();
yourCar.sManufacturer = "Rambler";
// crée une nouvelle référence et la fait pointer vers le même objet
Vehicle yourSpousalCar = yourCar;
```

La première ligne crée un objet `yourCar` sans lui assigner une valeur. On dit qu'une référence qui n'a pas été initialisée pointe vers l'*objet* `null`. Toute tentative d'utiliser une référence non initialisée produit une erreur immédiate qui met fin à l'exécution du programme.



Le compilateur C# est capable d'identifier la plupart des tentatives d'utiliser une référence non initialisée, et d'afficher un avertissement lors de la génération. Si une telle erreur n'est pas détectée par le compilateur, tout accès à une référence non initialisée met fin immédiatement à l'exécution du programme.

La deuxième instruction crée un nouvel objet `Vehicle`, et l'assigne à `yourCar`. La dernière instruction de ce fragment de code assigne la référence `yourSpousalCar` à la référence `yourCar`. Comme le montre la Figure 6.1, le résultat de cette instruction est que `yourSpousalCar` se réfère au même objet que `yourCar`.

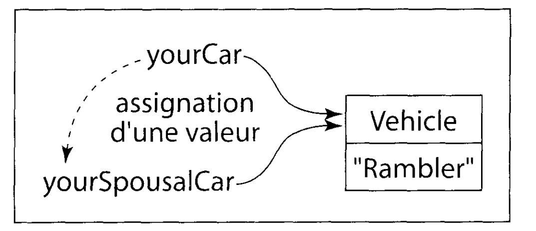


Figure 6.1 : La relation entre deux références au même objet.

Les deux appels suivants ont le même effet :

```
// construisez votre voiture
Vehicle yourCar = new Vehicle();
yourCar.sModel = "Ford T";
// elle appartient aussi à votre femme
Vehicle yourSpousalCar = yourCar;
// si l'une change, l'autre change aussi
yourSpousalCar.sModel = "Daytona";
Console.WriteLine("votre voiture est une " + yourCar.sModel);
```

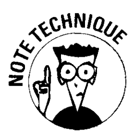
L'exécution de ce programme afficherait "Daytona", et non "Ford T". Remarquez que `yourSpousalCar` ne pointe pas vers `yourCar`. Au contraire, ce sont les deux qui se réfèrent au même véhicule.

En outre, la référence `yourSpousalCar` serait encore valide, même si la variable `yourCar` était d'une manière ou d'une autre "perdue" (se trouvait hors de portée, par exemple) :

```
// construisez votre voiture
Vehicle yourCar = new Vehicle();
yourCar.sModel = "Ford T";
// elle appartient aussi à votre femme
```

```
Vehicle yourSpousalCar = yourCar;
// quand elle s'en va avec votre voiture. . .
yourCar = null;         // yourCar référence maintenant "l'objet NULL"
//  . . .yourSpousalCar référence toujours le même véhicule
Console.WriteLine("Votre voiture était une " + yourSpousalCar.sModel);
```

L'exécution de ce programme affiche le résultat "Votre voiture était une Ford T", bien que la référence à yourCar ne soit plus valide.



L'objet n'est plus *accessible* à partir de la référence yourCar. Il ne devient pas complètement inaccessible tant que yourCar et yourSpousalCar ne sont pas "perdus" ou annulés.

Les classes qui contiennent des classes sont les plus heureuses du monde

Les membres d'une classe peuvent eux-mêmes être des références à d'autres classes. Par exemple, un véhicule a un moteur, qui a une puissance et différents paramètres qui définissent son efficacité (mais un vélo n'a pas de cylindrée). On peut introduire ces différents facteurs dans la classe, comme ceci :

```
public class Vehicle
{
  public string sModel;         // nom du modèle
  public string sManufacturer;  // nom du constructeur
  public int nNumOfDoors;       // nombre de portes du véhicule
  public int nNumOfWheels;      // nombre de roues du véhicule
  public int nPower;            // puissance du moteur (Chevaux-Vapeur)
  public double displacement;   // cylindrée du moteur (litres)
}
```

Toutefois, la puissance et la cylindrée du moteur ne résument pas toutes les caractéristiques de la voiture. Par exemple, la Jeep de mon fils est proposée avec deux moteurs différents qui lui donnent une puissance complètement différente. La Jeep de 2,4 litres de cylindrée est un veau, alors que la même voiture équipée du moteur de 4 litres est plutôt nerveuse.

Autrement dit, le moteur est une entité à lui seul et mérite sa propre classe :

```
class Motor
{
  public int nPower;            // puissance du moteur (Chevaux-Vapeur)
```

```
  public double displacement;    // cylindrée du moteur (litres)
}
```

Et vous pouvez utiliser cette classe dans la classe `Vehicle` :

```
public class Vehicle
{
  public string sModel;          // nom du modèle
  public string sManufacturer;   // nom du constructeur
  public int nNumOfDoors;        // nombre de portes du véhicule
  public int nNumOfWheels;       // nombre de roues du véhicule
  public Motor motor;
}
```

La création de `myCar` se présente maintenant ainsi :

```
//créons d'abord un objet de la classe Motor
Motor largerMotor = new Motor();
largerMotor.nPower = 230;
largerMotor.displacement = 4.0;
//créons maintenant la voiture
Vehicle sonsCar = new Vehicle();
sonsCar.sModel = "Cherokee Sport";
sonsCar.sManfacturer = "Jeep";
sonsCar.nNumberofDoors = 2;
sonsCar.numberOfWheels = 4;
//mettons un moteur dans la voiture
sonsCar.motor = largerMotor;
```

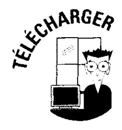
L'objet de la classe `Vehicle` vous offre deux moyens d'accéder à la cylindrée de son moteur. Vous pouvez procéder une étape à la fois :

```
Motor m = sonsCar.motor;
Console.WriteLine("La cylindrée du moteur est " + m.displacement);
```

Ou alors, y accéder directement :

```
Console.Writeline("La cylindrée du moteur est " + sonsCar.motor.displacement);
```

D'une manière ou d'une autre, vous ne pouvez accéder à la cylindrée (`displacement`) que par l'objet de la classe `Motor`.



Cet exemple fait partie du programme `VehicleAndMotor` qui se trouve sur le site Web.

Les membres statiques d'une classe

La plupart des membres d'une classe servent à décrire chaque objet de cette classe. Voyez la classe `Car` :

```
public class Car
{
  public string sLicensePlate;     //le numéro d'immatriculation
}
```

Le numéro d'immatriculation est une *propriété d'objet*, ce qui signifie qu'il définit individuellement chaque objet de la classe `Car`. Par exemple, vous avez de la chance que ma voiture n'ait pas le numéro d immatriculation que la vôtre. Ça pourrait vous attirer des ennuis.

```
Car myCar = new Car();
myCar.sLicensePlate = "XYZ123";

Car yourCar = new Car();
yourCar.sLicensePlate = "ABC789";
```

Mais il y a aussi des propriétés partagées par toutes les voitures. Par exemple, le nombre total de voitures construites est une propriété de la classe `Car`, et non de quelconque objet. Un tel membre d'une classe est appelé *propriété de classe*, et est identifié en C# par le mot `static` :

```
public class Car
{
  public static int nNumberOfCars; //nombre de voitures construites
  public string sLicensePlate;     //le numéro d'immatriculation
}
```

Ce n'est pas par un objet de la classe qu'on accède à un membre statique, mais par la classe elle-même, comme le montre cet exemple :

```
// crée un nouvel objet de la classe Car
Car newCar = new Car();
newCar.sLicensePlate = "ABC123";
// incrémente le nombre de voitures pour tenir compte de la nouvelle
Car.nNumberOfCars++;
```



On accède au membre d'objet `newCar.sLicensePlate` par l'objet `newCar`, alors qu'on accède au membre de classe (statique) `Car.nNumberOfCars` par la classe `Car`.

Définir des membres de type const

Le type const est un type spécial de membre statique. La valeur d'une variable const doit être établie dans la déclaration, et vous ne pouvez la changer nulle part dans le programme :

```
class Class1
{
  // nombre de jours dans l'année
  public const int nDaysInYear = 366;
  public static void Main(string[] args)
  {
    int[] nMaxTemperatures = new int[nDaysInYear];
    for(int index = 0; index < nDaysInYear; index++)
    {
      //  . . .additionne la température maximale pour chaque
      // jour de l'année. . .
    }
  }
}
```

Vous pouvez utiliser en n'importe quel endroit de votre programme la constante nDaysInYear à la place de la valeur 366. L'utilité d'une variable de type const est de remplacer une constante dépourvue de signification telle que 366 par un nom descriptif comme nDaysInYear, ce qui rend le programme plus lisible.

Les tableaux : la classe Array

Les variables qui ne contiennent qu'une seule valeur sont bien pratiques, et les classes qui permettent de décrire les objets composites sont cruciales. Mais il vous faut aussi une structure capable de contenir un ensemble d'objets de même type. La classe intégrée Array est une structure qui peut contenir une série d'éléments de même type (valeurs de type int, double, et ainsi de suite, ou alors objets de la classe Vehicle, Motor, et ainsi de suite).

Les arguments du tableau

Considérez le problème du calcul de la moyenne d'un ensemble de dix nombres en virgule flottante. Chacun de ces dix nombres nécessite son

propre stockage au format `double` (calculer une moyenne avec des variables `int` pourrait produire des erreurs d'arrondi, comme nous l'avons dit au Chapitre 3) :

```
double d0 = 5;
double d1 = 2;
double d2 = 7;
double d3 = 3.5;
double d4 = 6.5;
double d5 = 8;
double d6 = 1;
double d7 = 9;
double d8 = 1;
double d9 = 3;
```

Vous devez maintenant faire la somme de toutes ces valeurs, puis la diviser par 10 (le nombre de valeurs) :

```
double dSum = d0 + d1 + d2 + d3 + d4 + d5 + d6 + d7 + d8 + d9;
double dAverage = dSum / 10;
```

Il est un peu fastidieux d'écrire le nom de chacun de ces éléments pour en faire la somme. Passe encore si vous n'avez que dix nombres, mais imaginez que vous en ayez cent ou même mille.

Le tableau à longueur fixe

Heureusement, vous n'avez pas besoin de nommer chaque élément séparément. C# offre une structure, le tableau, qui permet de stocker une séquence de valeurs. En utilisant un tableau, vous pouvez réécrire de la façon suivante le premier fragment de code de la section précédente :

```
double[] dArray = {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
```



La classe `Array` présente une syntaxe spéciale qui la rend plus facile à utiliser. Les doubles crochets `[]` représentent la manière qui permet d'accéder aux différents éléments du tableau :

```
dArray[0] correspond à d0
dArray[1] correspond à d1
. . .
```

L'élément numéro 0 du tableau correspond à d0, l'élément numéro 1 à d1, et ainsi de suite.

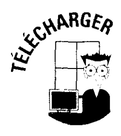


Les numéros des éléments du tableau (0, 1, 2, et ainsi de suite) constituent l'*index*.



L'index d'un tableau commence à 0, et non à 1. Par conséquent, l'élément du tableau correspondant à l'index 1 n'est pas le premier élément, mais l'élément numéro 1, ou "l'élément 1 de l'index". Le premier élément est l'élément numéro 0. Si vous voulez vraiment parler normalement, souvenez-vous que le premier élément est à l'index 0, et le deuxième à l'index 1.

dArray ne constituerait pas une grande amélioration sans la possibilité d'utiliser une variable comme index du tableau. Il est plus facile d'utiliser une boucle for que de se référer manuellement à chaque élément, comme le montre le programme suivant :




```
// FixedArrayAverage - calcule la moyenne d'un nombre déterminé
//                     de valeurs en utilisant une boucle
namespace FixedArrayAverage
{
  using System;
  public class Class1
  {
    public static int Main(string[] args)
    {
      double[] dArray =
              {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
      // fait la somme des valeurs du tableau
      // dans la variable dSum
      double dSum = 0;
      for (int i = 0; i < 11; i++)
      {
        dSum = dSum + dArray[i];
      }
      // calcule maintenant la moyenne
      double dAverage = dSum / 10;
      Console.WriteLine(dAverage);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
}
```

Le programme commence par initialiser à 0 la variable `dSum`. Il effectue ensuite une boucle sur les valeurs stockées dans `dArray`, en ajoutant chacune d'elles à `dSum`. À la fin de la boucle, `dSum` contient la somme de toutes les valeurs du tableau. Celle-ci est alors divisée par le nombre d'éléments pour obtenir la moyenne. Le résultat affiché par l'exécution de ce programme est 4,6, comme on pouvait l'attendre (j'ai vérifié avec ma calculatrice).

Et si vous dépassez la taille du tableau ?

Le programme `FixedArrayAverage` effectue une boucle sur un tableau de dix éléments. Heureusement, cette boucle passe effectivement sur tous ces dix éléments. Mais si j'avais fait une erreur dans l'itération ? Il y a deux cas à envisager.

Si je n'avais itéré que sur neuf éléments, C# ne l'aurait pas considéré comme une erreur : si vous voulez lire neuf éléments d'un tableau qui en contient dix, de quel droit C# viendrait-il le contester ? Bien sûr, la moyenne serait incorrecte, mais le programme n'aurait aucun moyen de le savoir.

Et si j'avais itéré sur onze éléments (ou plus) ? Maintenant, ça regarde beaucoup C#. C# ne vous permet pas d'indexer au-delà de la taille d'un tableau, de crainte d'écraser une valeur importante dans la mémoire. Pour le vérifier, j'ai remplacé le test de comparaison de la boucle `for` par ce qui suit : `for(int i = 0; i < 11; i++)`, en remplaçant 10 par 11. L'exécution du programme a produit l'erreur suivante (en français et en anglais dans le texte) :

```
Exception non gérée : System.IndexOutOfRangeException : une
exception qui de type System.IndexOutOfRangeException a été levée.
  at FixedArrayAverage.Class1.Main(String[] args) in c:\c#programs\
  fixedarrayaverage\class1.cs:line 17
```

Au premier abord, ce message d'erreur paraît imposant, mais on peut facilement en saisir l'essentiel : il s'est produit une erreur `IndexOutOfRangeException`. Il est clair que C# indique que le programme a essayé d'accéder à un tableau au-delà de ses limites (le onzième élément d'un tableau qui n'en comporte que dix). La suite du message indique la ligne exacte à laquelle s'est produite cette tentative d'accès, mais vous n'avez pas encore assez avancé dans ce livre pour comprendre tout ce qu'il vous dit.

```
      // déclare un tableau de la taille correspondante
      double[] dArray = new double[numElements];
      // remplit le tableau avec les valeurs
      for (int i = 0; i < numElements; i++)
      {
        // demande à l'utilisateur une nouvelle valeur
        Console.Write("Entrez la valeur nº" + (i + 1) + ": ");
        string sVal = Console.ReadLine();
        double dValue = Convert.ToDouble(sVal);
        // stocke la nouvelle valeur dans le tableau
        dArray[i] = dValue;
      }
      // fait l'addition de 'numElements' valeurs
      // du tableau dans la variable dSum
      double dSum = 0;
      for (int i = 0; i < numElements; i++)
      {
        dSum = dSum + dArray[i];
      }
      // calcule maintenant la moyenne
      double dAverage = dSum / numElements;
      // affiche les résultats dans un format agréable
      Console.WriteLine();
      Console.Write(dAverage
                  + " est la moyenne de ("
                  + dArray[0]);
      for (int i = 1; i < numElements; i++)
      {
        Console.Write(" + " + dArray[i]);
      }
      Console.WriteLine(") / " + numElements);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
}
```

Voici un exemple de résultats affichés, pour lequel j'ai entré cinq valeurs, de 1 à 5, dont la moyenne calculée par le programme est de 3 :

```
Nombre de valeurs pour la moyenne à calculer : 5

Entrez la valeur nº1: 1
Entrez la valeur nº2: 2
Entrez la valeur nº3: 3
Entrez la valeur nº4: 4
```

```
{
  // demande à l'utilisateur une nouvelle valeur
  Console.Write("Entrez la valeur nº" + (i + 1) + ": ");
  string sVal = Console.ReadLine();
  double dValue = Double.FromString(sVal);
  // stocke la nouvelle valeur dans le tableau
  dArray[i] = dValue;
}
```

Le tableau `dArray` est déclaré comme ayant une longueur de `numElements`. L'astucieux programmeur (moi-même) a donc utilisé une boucle `for` pour itérer `numElements` fois sur les éléments du tableau.

Il serait lamentable d'avoir à trimbaler partout avec `dArray` la variable `numElements`, rien que pour connaître la longueur du tableau. Heureusement, ce n'est pas nécessaire. Un tableau possède une propriété nommée `Length` qui contient sa longueur. `dArray.Length` a donc la même valeur que `numElements`.

La boucle `for` suivante aurait été préférable :

```
// remplit le tableau avec les valeurs
for (int i = 0; i < dArray.Length; i++)
{
```

Pourquoi les déclarations des tableaux de longueur fixe et de longueur variable sont-elles si différentes ?

Superficiellement, la syntaxe de la déclaration d'un tableau de longueur fixe ou de longueur variable est assez différente :

```
double[] dFixedLengthArray = {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
double[] dVariableLengthArray = new double[10];
```

La différence vient du fait que C# essaie de vous éviter un peu de travail. C# alloue à votre place la mémoire nécessaire dans le cas d'un tableau de longueur fixe comme `dFixedLengthArray`. J'aurais pu aussi le faire moi-même :

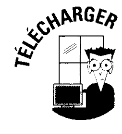
```
double[] dFixedLengthArray = new double[10] {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
```

Ici, j'ai utilisé `new` pour allouer explicitement la mémoire, et j'ai fait suivre cette déclaration par les valeurs initiales des membres du tableau.

Le programme peut maintenant définir les propriétés de chaque étudiant :

```
students[i] = new Student();
students[i].sName = "Mon nom";
students[i].dGPA = dMyGPA;
```

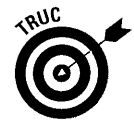
C'est cette merveille que vous pouvez voir dans le programme AverageStudentGPA ci-dessous, qui recueille des informations sur un certain nombre d'étudiants et affiche la moyenne globale des points de leurs unités de valeur :




```
// AverageStudentGPA - calcule la moyenne des points
//                     d'UV (GPA)d'un certain nombre d'étudiants.
using System;
namespace AverageStudentGPA
{
  public class Student
  {
    public string sName;
    public double dGPA;         // moyenne des points d'UV
  }
  public class Class1
  {
    public static void Main(string[] args)
    {
      // demande le nombre d'étudiants
      Console.WriteLine("Entrez le nombre d'étudiants");
      string s = Console.ReadLine();
      int nNumberOfStudents = Convert.ToInt32(s);
      // définit un tableau d'objets Student
      Student[] students = new Student[nNumberOfStudents];
      // remplit maintenant le tableau
      for (int i = 0; i < students.Length; i++)
      {
        // demande le nom à l'utilisateur, et ajoute 1
        // à l'index, parce que les objets des tableaux en C#
        // sont numérotés à partir de 0
        Console.Write("Entrez le nom de l'étudiant "
                      + (i + 1) + ": ");
        string sName = Console.ReadLine();
        Console.Write("Entrez sa moyenne de points d'UV : ");
        string sAvg = Console.ReadLine();
        double dGPA = Convert.ToDouble(sAvg);
        // crée un objet Student à partir de ces données
        Student thisStudent = new Student();
        thisStudent.sName = sName;
```

```
Entrez sa moyenne de points d'UV : 3.5
Entrez le nom de l'étudiant 3: Carrie
Entrez sa moyenne de points d'UV : 4.0

La moyenne générale des 3 étudiants est 3.5
Appuyez sur Entrée pour terminer...
```



Le nom d'une variable de référence à des objets doit de préférence être au singulier, comme `student`. D'autre part, il doit inclure le nom de la classe, comme dans `badStudent` ou `goodStudent`, ou encore `sexyCoedStudent`. Le nom d'un tableau (ou de tout autre collection, à vrai dire) doit de préférence être au pluriel, comme `students` ou `phoneNumbers`, ou encore `phoneNumbersInMyPalmPilot`. Comme d'habitude, ces suggestions ne reflètent que l'opinion de l'auteur de ce livre et non de l'éditeur, encore moins de ses actionnaires. C# ne se préoccupe absolument pas de la manière dont vous définissez les noms de vos variables.

Une structure de contrôle de flux pour tous les tableaux : `foreach`

À partir d'un tableau d'objets de la classe `Student`, la boucle suivante calcule la moyenne des points de leurs UV :

```
public class Student
{
  public string sName;
  public double dGPA;         // moyenne des points d'UV
}
public class Class1
{
  public static void Main(string[] args)
  {
    //  . . .crée le tableau. . .
    // et fait la moyenne des étudiants du tableau
    double dSum = 0.0;
    for (int i = 0; i < students.Length; i++)
    {
      dSum += students[i].dGPA;
    }
    double dAvg = dSum/students.Length;
    //  . . .utilise le tableau. . .
  }
}
```



La boucle `for` effectue une itération sur tous les membres du tableau.

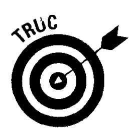
`students.Length` contient le nombre d'éléments du tableau.

C# offre une structure de contrôle de flux, nommée `foreach`, spécialement conçue pour l'itération dans un conteneur tel qu'un tableau. Elle fonctionne de la façon suivante :

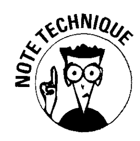
```
// fait la moyenne des étudiants du tableau
double dSum = 0.0;
foreach (Student stud in students)
{
  dSum += stud.dGPA;
}
double dAvg = dSum/students.Length;
```

Lors du premier passage de la boucle, l'instruction `foreach` va chercher le premier objet `Student` dans le tableau, et le stocke dans la variable `stud`. À chaque passage successif, l'instruction `foreach` va chercher l'élément suivant. Le contrôle sort de `foreach` lorsque tous les éléments du tableau ont été traités de cette façon.

Remarquez qu'aucun index n'apparaît dans l'instruction `foreach`, ce qui réduit considérablement les risques d'erreur.



Les programmeurs C, C++ et Java ne se sentiront sans doute pas très confortables au premier abord avec `foreach`, car cette instruction est propre à C#. Mais elle a la qualité de grandir en quelque sorte pour vous. Pour accéder aux tableaux, c'est la plus facile à utiliser de toutes les commandes de boucle.



La structure `foreach` est en fait plus puissante que cet exemple ne le laisse paraître. En dehors des tableaux, elle fonctionne aussi avec d'autres types de collection (les collections sont expliquées au Chapitre 16). D'autre part, notre exemple de `foreach` ignorerait les éléments du tableau qui ne seraient pas du type `Student`.

Trier un tableau d'objets

La nécessité de trier les éléments d'un tableau est une difficulté bien connue de la programmation. Que la taille d'un tableau ne puisse être ni augmentée ni réduite ne signifie pas que ces éléments ne puissent pas

être déplacés, ni que l'on ne puisse en ajouter ou en supprimer. Par exemple, le fragment de code suivant permute deux éléments de `Student` dans le tableau `students` :

```
Student temp = students[i]; // met de côté l'étudiant n°i
students[i] = students[j];
students[j] = temp;
```

Ici, la référence d'objet de l'emplacement `i` du tableau `students` est sauvegardé, afin qu'elle ne soit pas perdue lorsque la deuxième instruction change la valeur de `students[i]`. Enfin, le contenu de la variable `temp` est stocké à l'emplacement j. Dans une représentation visuelle, cette opération ressemble à la Figure 6.2.

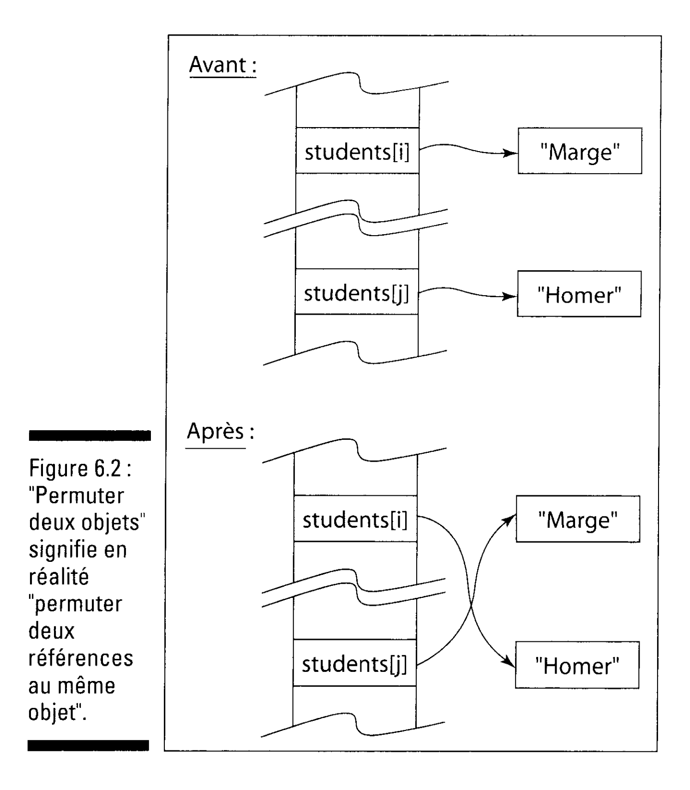
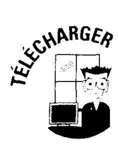


Figure 6.2 : "Permuter deux objets" signifie en réalité "permuter deux références au même objet".

Le programme suivant donne une démonstration de la manière d'utiliser la possibilité de manipuler les éléments d'un tableau pour effectuer un tri. Cet algorithme de tri s'appelle "en bulles" (*bubble sort*). Ce n'est pas le

meilleur sur de grands tableaux contenant des milliers d'éléments, mais il est simple et efficace pour de petits tableaux :




```
// SortStudents - ce programme montre comment trier
//                un tableau d'objets
using System;
namespace SortStudents
{
  class Class1
  {
    public static void Main(string[] args)
    {
      // crée un tableau d'étudiants
      Student[] students = new Student[5];
      students[0] = Student.NewStudent("Homer", 0);
      students[1] = Student.NewStudent("Lisa", 4.0);
      students[2] = Student.NewStudent("Bart", 2.0);
      students[3] = Student.NewStudent("Marge", 3.0);
      students[4] = Student.NewStudent("Maggie", 3.5);
      // output the list as is:
      Console.WriteLine("Avant de trier :");
      OutputStudentArray(students);
      // trie maintenant la liste des étudiants
      // du meilleur au plus mauvais
      Console.WriteLine("\nTri en cours\n");
      Student.Sort(students);
      // affiche la liste triée
      Console.WriteLine("Étudiants par résultats décroissants :");
      OutputStudentArray(students);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // OutputStudentArray - affiche tous les étudiants du tableau
    public static void OutputStudentArray(Student[] students)
    {
      foreach(Student s in students)
      {
        Console.WriteLine(s.GetString());
      }
    }
  }
  // Student - description d'un étudiant (nom et résultats)
  class Student
  {
    public string sName;
    public double dGrade = 0.0;
```

```
// NewStudent - retourne un nouvel objet Student initialisé
public static Student NewStudent(string sName, double dGrade)
{
  Student student = new Student();
  student.sName = sName;
  student.dGrade = dGrade;
  return student;
}
// GetString - convertit en chaîne l'objet Student
//             en cours
public string GetString()
{
  string s = "";
  s += dGrade;
  s += " - ";
  s += sName;
  return s;
}
// Sort - trie un tableau d'étudiants par ordre de résultats
//        décroissants, avec l'algorithme en bulles
public static void Sort(Student[] students)
{
  bool bRepeatLoop;
  // répète la boucle jusqu'à ce que la liste soit triée
  do
  {
    // cet indicateur sera défini comme vrai si un objet
    // est trouvé mal classé
    bRepeatLoop = false;
    // itère sur la liste des étudiants
    for(int index = 0; index < (students.Length - 1); index++)
    {
      // si deux étudiants sont dans le mauvais sens. . .
      if (students[index].dGrade <
                      students[index + 1].dGrade)
      {
        // . . .ils sont permutés. . .
        Student to = students[index];
        Student from = students[index + 1];
        students[index]     = from;
        students[index + 1] = to;
        // . . .et l'indicateur bRepeatLoop dit si il faudra
        // faire encore un passage sur la liste des étudiants
        // (continue à itérer jusqu'à ce que tous les objets
        // soient dans le bon ordre)
        bRepeatLoop = true;
      }
    }
```

```
      } while (bRepeatLoop);
    }
  }
}
```

Commençons par examiner ce qu'affiche le programme, rien que pour nous faire une idée :

```
Avant de trier :
0 - Homer
4 - Lisa
2 - Bart
3 - Marge
3.5 - Maggie

Tri en cours

Étudiants par résultats décroissants :
4 - Lisa
3.5 - Maggie
3 - Marge
2 - Bart
0 - Homer
Appuyez sur Entrée pour terminer...
```

Afin de gagner du temps, le vôtre comme le mien, j'ai codé localement la création de cinq étudiants. La méthode `NewStudent()` alloue un nouvel objet `Student`, initialise son nom et son "grade", et retourne le résultat. Le programme utilise la fonction `OutputStudentArray()` pour afficher le tableau des étudiants avant qu'il soit trié.

Le programme invoque ensuite la fonction `Sort()`.

Après le tri, le programme répète le processus d'affichage dans le seul but de vous impressionner avec le résultat maintenant trié.

Bien sûr, la principale nouveauté du programme `SortStudents` est la méthode `Sort()`. Cet algorithme fonctionne en effectuant une boucle continue sur la liste des étudiants jusqu'à ce qu'elle soit triée. À chaque passage, le programme compare chaque étudiant à son voisin. S'ils ne sont pas dans le bon ordre, la fonction les permute et met à jour un indicateur pour signaler que la liste n'était pas complètement triée. Les Figures 6.3 à 6.6 montrent la liste des étudiants après chaque passage.

Figure 6.3 : Avant de commencer le tri en bulles.

Homer	Ø
Lisa	4
Bart	2
Marge	3
Maggie	3.5

Figure 6.4 : Après le premier passage du tri en bulles.

Lisa	4	
Bart	2	
Marge	3	
Maggie	3.5	
Homer	Ø	← Homer finit par se retrouver tout en bas.

Figure 6.5 : Après le deuxième passage du tri en bulles.

Lisa	4	← Lisa reste tout en haut.
Marge	3	
Maggie	3.5	
Bart	2	← Bart est descendu, mais reste au-dessus d'Homer.
Homer	Ø	

Figure 6.6 : Après l'avant-dernier passage, la liste est triée. Le passage final met fin au tri en constatant que rien ne change.

Lisa	4	
Maggie	3.5	Maggie et Marge sont permutées.
Marge	3	
Bart	2	
Homer	Ø	

Au bout du compte, les meilleurs étudiants, comme Lisa et Maggie, remontent comme des bulles jusqu'en haut, alors que les plus mauvais, comme Homer, tombent au fond. Voilà pourquoi ça s'appelle le tri en bulles.



Ce qui permet de réaliser une fonction de tri, celle-ci ou n'importe quelle autre, c'est que les éléments du tableau peuvent être réordonnés en assignant à un autre élément du tableau la valeur qui en référence un autre. Remarquez que l'assignation d'une référence ne fait pas une copie de l'objet, raison pour laquelle c'est une opération très rapide.

Chapitre 7

Mettre en marche quelques fonctions de grande classe

Dans ce chapitre :

Définir une fonction.

Passer des arguments à une fonction.

Obtenir des résultats (c'est agréable).

Étudier l'exemple `WriteLine()`.

Passer des arguments au programme.

Les programmeurs ont besoin d'avoir la possibilité de diviser de grands programmes en morceaux plus petits, donc plus faciles à manier. Par exemple, les programmes présentés dans les chapitres précédents sont proches des limites de ce qu'une personne normalement constituée peut digérer en une seule fois.

C# permet de diviser le code d'un programme en un certain nombre de morceaux que l'on appelle des *fonctions*. Des fonctions bien conçues et bien implémentées peuvent simplifier considérablement le travail d'écriture d'un programme complexe.

Définir et utiliser une fonction

Considérez l'exemple suivant :

```
class Example
{
```

```
    public int nInt;
    public static int nStaticInt
    public void MemberFunction()
    {
      Console.WriteLine("ceci est une fonction membre");
    }
    public static void ClassFunction()
    {
      Console.WriteLine("ceci est une fonction de classe");
    }
  }
```

L'élément nInt est un *membre donnée*, une donnée membre d'une classe (la classe Example), comme nous en avons vu beaucoup au Chapitre 6, mais l'élément MemberFunction() est d'un genre nouveau : c'est une *fonction membre* (une fonction, membre d'une classe). Une fonction est un bloc de code C# que vous pouvez exécuter en référençant son nom. Ce sera plus clair avec un exemple (n'oubliez pas nos conventions, le nom d'une classe commence par une majuscule : example est un objet de la classe Example).

Le fragment de code suivant assigne une valeur à la donnée nInt (membre de l'objet example de la classe Example), et à la variable statique nStaticInt (référencée par la classe Example et non par un objet de cette classe, puisqu'elle est statique) :

```
Example example = new Example(); //crée un objet
example.nInt = 1;                //utilise l'objet pour initialiser
                                 //un membre donnée
Example.nStaticInt = 2;          //utilise la classe pour initialiser
                                 //un membre statique
```

Le fragment de code suivant définit et invoque MemberFunction() et ClassFunction(), presque de la même manière :

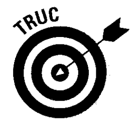
```
Example example = new Example(); //crée un objet
example.MemberFunction();        //utilise l'objet pour invoquer
                                 //une fonction membre
Example.ClassFunction();         //utilise la classe pour invoquer
                                 //une fonction de classe
```

L'expression example.MemberFunction() passe le contrôle au code contenu dans la fonction. Le processus suivi par C# pour Example.ClassFunction() est presque identique. L'exécution de ce simple fragment de code produit

l'affichage suivant à l'aide de l'instruction `WriteLine()` contenue dans chaque fonction :

```
ceci est une fonction membre
ceci est une fonction de classe
```

Une fois que la fonction a achevé son exécution, elle restitue le contrôle au point où elle a été invoquée.

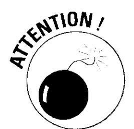


Lorsque je décris des fonctions, je mets les parenthèses pour qu'elles soient plus faciles à reconnaître, sinon j'ai un peu de mal à m'y retrouver.

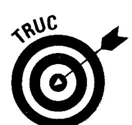
Ce petit morceau de code C# avec ses deux exemples de fonctions ne fait rien d'autre qu'afficher deux petites chaînes de caractères sur la console, mais une fonction effectue généralement des opérations utiles et parfois complexes ; par exemple calculer un sinus ou concaténer deux chaînes de caractères, ou encore, envoyer subrepticement par mail votre URL à Microsoft. Vous pouvez faire des fonctions aussi longues et compliquées que vous voulez.

Un exemple de fonction pour vos fichiers

Dans cette section, je vais reprendre les exemples monolithiques du programme `CalculateInterestTable` du Chapitre 5, et les diviser en plusieurs fonctions de taille raisonnable, pour montrer à quel point l'utilisation de fonctions peut contribuer à rendre le programme plus facile à écrire et à comprendre.



Je décrirai en détail les manières de définir et d'appeler une fonction dans des sections ultérieures de ce chapitre. Cet exemple n'est là que pour donner une vue d'ensemble.



En lisant simplement les commentaires sans le code C#, vous devez pouvoir vous faire une idée assez claire de ce que fait un programme. Si ce n'est pas le cas, c'est que les commentaires sont mal faits.

Dans les grandes lignes, le programme `CalculateInterestTable` apparaît comme suit :

```
public static void Main(string[] args)
{
  //demande à l'utilisateur d'entrer le principal initial
```

```
  //si le principal est négatif
  //génère un message d'erreur
  //demande à l'utilisateur d'entrer le taux d'intérêt
  //si le taux d'intérêt est négatif, génère un message d'erreur
  //demande à l'utilisateur d'entrer le nombre d'années
  //affiche les données saisies par l'utilisteur
  //effectue une boucle avec le nombre d'années spécifié
  while(nYear <= nDuration)
  {
    //calcule la valeur du principal
    //plus l'intérêt
    //affiche les résultats
  }
}
```

Si vous l'examinez avec un peu de recul, vous verrez que ce programme se décompose en trois sections distinctes :

- Une section initiale de saisie dans laquelle l'utilisateur entre les données, à savoir le principal, le taux d'intérêt, et la durée.
- Une section d'affichage des données entrées, afin que l'utilisateur puisse les vérifier.
- Une section finale qui calcule et affiche les résultats.

Ce sont de bons endroits où regarder pour trouver la bonne manière de diviser un programme. En fait, si vous examinez de plus près la section de saisie de ce programme, vous pouvez voir que c'est essentiellement le même code qui est utilisé pour saisir :

- Le principal.
- Le taux d'intérêt.
- La durée.

Cette observation nous donne un autre bon endroit où chercher.

C'est à partir de là que j'ai créé le programme `CalculateInterestTable-WithFunctions` :




```
// CalculateInterestTableWithFunctions - génère une table d'intérêts
//                                 semblable à d'autres programmes de
//                                 table d'intérêts, mais utilise une
//                                 certaine division du travail
//                                 entre plusieurs fonctions.
```

```
using System;
namespace CalculateInterestTableWithFunctions
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      //Section 1 - saisie des données nécessaires pour créer la table
      decimal mPrincipal = 0;
      decimal mInterest = 0;
      decimal mDuration = 0;
      InputInterestData(ref mPrincipal,
                        ref mInterest,
                        ref mDuration);
      //Section 2 - affiche les données pour vérification
      Console.WriteLine();  // skip a line
      Console.WriteLine("Principal      = " + mPrincipal);
      Console.WriteLine("Taux d'intérêt = " + mInterest + "%");
      Console.WriteLine("Durée          = " + mDuration + " ans");
      Console.WriteLine();
      //Section 3 - affiche la table des intérêts calculés
      OutputInterestTable(mPrincipal, mInterest, mDuration);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // InputInterestData - lit à partir du clavier le principal,
    //                     les informations sur le taux d'intérêt
    //                     et la durée, nécessaires pour calculer
    //                     la table des valeurs futures
    //(Cette fonction implémente la Section 1 en la divisant
    //en trois composants)
    public static void InputInterestData(ref decimal mPrincipal,
                                         ref decimal mInterest,
                                         ref decimal mDuration)
    {
      // 1a - lecture du principal
      mPrincipal = InputPositiveDecimal("le principal");
      // 1b - lecture du taux d'intérêt
      mInterest = InputPositiveDecimal("le taux d'intérêt");
      // 1c - lecture de la durée
      mDuration = InputPositiveDecimal("la durée");
    }
    // InputPositiveDecimal - lecture d'un nombre décimal positif
    //                        à partir du clavier
    //(saisie du principal, du taux d'intérêt ou de la durée
    //il s'agit de saisir un nombre décimal et de
    //vérifier qu'il est positif)
```

```
    public static decimal InputPositiveDecimal(string sPrompt)
    {
      // continue jusqu'à ce que l'utilisateur entre une valeur valide
      while(true)
      {
        // demande une valeur à l'utilisateur
        Console.Write("Entrez " + sPrompt + ":");
        // lit une valeur décimale saisie au clavier
        string sInput = Console.ReadLine();
        decimal mValue = Convert.ToDecimal(sInput);
        // sort de la boucle si la valeur entrée est correcte
        if (mValue >= 0)
        {
          // retourne la valeur décimale valide entrée par l'utilisateur
          return mValue;
        }
        // sinon, génère un message pour signaler l'erreur
        Console.WriteLine(sPrompt + " doit avoir une valeur positive");
        Console.WriteLine("Veuillez recommencer");
        Console.WriteLine();
      }
    }
    // OutputInterestTable - à partir du principal et du taux d'intérêt,
    //                       génère la table des valeurs futures pour
    //                       le nombre de périodes indiquées par
    //                       mDuration.
    //(ceci implémente la section 3 du programme)
    public static void OutputInterestTable(decimal mPrincipal,
                                           decimal mInterest,
                                           decimal mDuration)
    {
      for (int nYear = 1; nYear <= mDuration; nYear++)
      {
        // calcule la valeur du principal
        // plus l'interêt
        decimal mInterestPaid;
        mInterestPaid = mPrincipal * (mInterest / 100);
        // calcule maintenant le nouveau principal en ajoutant
        // l'intérêt au précédent principal
        mPrincipal = mPrincipal + mInterestPaid;
        // arrondit le principal au centime le plus proche
        mPrincipal = decimal.Round(mPrincipal, 2);
        // affiche le résultat
        Console.WriteLine(nYear + "-" + mPrincipal);
      }
    }
  }
}
```

J'ai divisé l'ensemble `Main()` en trois parties clairement distinctes, dont chacune est indiquée par un commentaire en gras, ensuite j'ai divisé à nouveau la première section en 1a, 1b, et 1c.



Normalement, les commentaires en gras n'auraient rien à faire là. Ils ne feraient qu'encombrer inutilement le source. En pratique, de tels commentaires sont inutiles si vos fonctions sont bien conçues.

La section 1 appelle la fonction `InputInterestData()` afin de saisir les trois variables dont le programme a besoin pour calculer les résultats : `mPrincipal`, `mInterest`, et `mDuration`. La section 2 affiche ces trois valeurs de la même manière que les versions antérieures du programme. La partie finale utilise la fonction `OutputInterestTable()` pour afficher les résultats.

Commençant par le bas et progressant vers le haut, la fonction `OutputInterestTable()` effectue une boucle pour calculer les intérêts successifs. C'est la même boucle que celle qui est utilisée dans le programme `CalculateInterestTable`, sans avoir recours à une fonction, au Chapitre 5. Mais l'avantage de cette version est que lorsque vous écrivez cette section du code, vous n'avez pas besoin de vous préoccuper des détails de la saisie et de la vérification des données. En écrivant cette fonction, vous avez simplement à penser : "Étant données trois valeurs, le principal, le taux d'intérêt et la durée, calculer et afficher la table des intérêts." C'est tout. Une fois que vous avez terminé, vous pouvez revenir à la ligne qui a invoqué la fonction `OutputInterestTable()`, et continuer à partir de là.

C'est la même logique de diviser pour régner qui est à l'œuvre dans la fonction `InputInterestData()`. Vous pouvez vous y concentrer exclusivement sur la saisie des trois valeurs décimales. Toutefois, dans ce cas, on s'aperçoit que la saisie de chaque valeur fait appel aux mêmes opérations. La fonction `InputPositiveDecimal()` rassemble ces opérations dans un bloc de code que vous pouvez appliquer tout aussi bien au principal, au taux d'intérêt, et à la durée.

Cette fonction `InputPositiveDecimal()` affiche l'invite qu'elle a reçue lorsqu'elle a été invoquée, et attend la saisie de l'utilisateur. Puis, si cette valeur n'est pas négative, elle la retourne au point où elle a été appelée. Si la valeur est négative, la fonction affiche un message d'erreur et demande à nouveau la valeur à l'utilisateur.

Du point de vue de l'utilisateur, ce programme fonctionne exactement de la même manière que la version monolithique du Chapitre 5, et c'est bien ce que nous voulions :

```
Entrez le principal:100
Entrez le taux d'intérêt:-10
le taux d'intérêt doit avoir une valeur positive
Veuillez recommencer

Entrez le taux d'intérêt:10
Entrez la durée:10

Principal      = 100
Taux d'intérêt = 10%
Durée          = 10 ans

1-110
2-121
3-133.1
4-146.41
5-161.05
6-177.16
7-194.88
8-214.37
9-235.81
10-259.39
Appuyez sur Entrée pour terminer...
```

J'ai donc pris un programme un peu long et un peu compliqué, et je l'ai divisé en éléments plus petits et plus compréhensibles, tout en faisant disparaître certaines duplications qu'il comportait.

Pourquoi des fonctions ?

Lorsque le langage FORTRAN a introduit la notion de fonction dans les années cinquante, le seul but en était d'éviter la duplication de code en rassemblant les portions identiques dans un seul élément commun. Imaginez que vous ayez eu à écrire un programme devant calculer et afficher des ratios en de nombreux endroits différents. Votre programme aurait pu appeler une fonction `DisplayRatio()` chaque fois que nécessaire, pratiquement dans le seul but de mettre plusieurs fois dans le programme le même bloc de code. L'économie peut paraître modeste pour une fonction aussi petite que `DisplayRatio()`, mais une fonction peut être beaucoup plus grande. En outre, une fonction d'usage courant comme `WriteLine()` peut être invoquée en des centaines d'endroits différents d'un même programme.

Il y a un autre avantage qui devient rapidement évident : il est plus facile d'écrire correctement le code d'une fonction. La fonction `DisplayRatio()` vérifie que le dénominateur n'est pas nul. Si vous répétez le code du calcul en plusieurs endroits dans votre programme, il est facile d'oublier ce test ici ou là.

Encore un avantage un peu moins évident : une fonction bien conçue réduit la complexité d'un programme. Une fonction bien définie doit correspondre à un concept bien défini. Il doit être possible d'en décrire la finalité sans utiliser les mots *et* et *ou*.

Une fonction comme `calculateSin()` constitue un exemple idéal. Le programmeur qui a besoin de réaliser de tels calculs peut alors implémenter cette opération complexe sans inquiétude sur la manière de l'utiliser, et sans se préoccuper de son fonctionnement interne. Le nombre de choses dont le programmeur doit se préoccuper en est considérablement réduit. D'autre part, en réduisant le nombre de "variables", une tâche importante se trouve réduite à deux tâches nettement plus petites et plus faciles.

Un programme de grande taille (par exemple un traitement de texte) se compose de nombreuses couches successives de fonctions, correspondant à des niveaux croissants d'abstraction. Par exemple, une fonction `RedisplayDocument()` appellerait sans aucun doute une fonction `Reparagraph()` pour réafficher les paragraphes dans le document. La fonction `Reparagraph()` devrait alors à son tour invoquer une fonction `CalculateWordWrap()` pour déterminer où placer les retours à la ligne qui déterminent l'affichage du paragraphe. `CalculateWordWrap()` elle-même appellerait une fonction `LookUpWordBreak()` pour décider des éventuelles coupures de mots à la fin des lignes. Comme vous le voyez, nous venons de décrire chacune de ces fonctions en une seule phrase simple.

Sans la possibilité de représenter des concepts complexes, il deviendrait presque impossible d'écrire des programmes de complexité simplement moyenne, a fortiori un système d'exploitation comme Windows XP, un utilitaire comme WinZip, un traitement de texte comme WordPerfect, ou encore un jeu comme StarFighter, pour ne citer que quelques exemples.

Donner ses arguments à une fonction

Une méthode comme celle de l'exemple suivant est à peu près aussi utile que ma brosse à cheveux car aucune donnée n'est passé à la fonction, et aucune n'en sort :

```
public static void Output()
{
  Console.WriteLine("ceci est une fonction");
}
```

Comparez cet exemple aux véritables fonctions qui font vraiment quelque chose. Par exemple, l'opération de calcul d'un sinus nécessite une donnée (il faut bien que ce soit le sinus de quelque chose). De même, pour concaténer deux chaînes en une seule, il faut commencer par en avoir deux. Il faut passer deux chaînes comme données à la fonction `Concatenate()`. Il vous faut donc un moyen de passer des données à une fonction et de récupérer ce qui en sort.

Passer un argument à une fonction

Les valeurs qui constituent des données d'une fonction sont appelées *arguments de la fonction*. La plupart des fonctions ont besoin d'arguments pour accomplir ce qu'elles ont à faire. Pour passer des arguments à une fonction, on en place la liste entre les parenthèses qui suivent son nom. Voyez maintenant la petite modification apportée à la classe `Example` :

```
public class Example
{
  public static void Output(string funcString)
  {
    Console.WriteLine("Output() a reçu l'argument : "
                     + funcString);
  }
}
```

J'aurais pu invoquer cette fonction depuis la même classe de la façon suivante :

```
Output("Hello");
```

Et j'aurais reçu le même mémorable résultat :

```
Output() a reçu l'argument : Hello
```

Le programme passe à la fonction `Output()` une référence à la chaîne "Hello". La fonction reçoit cet référence et lui assigne le nom `funcString`. La fonction `Output()` peut alors utiliser `funcString` dans le code qu'elle contient, comme n'importe quel autre variable de type `string`.

Je vais maintenant apporter une modification mineure à cet exemple :

```
string upperString = "Hello";
Output(upperString);
```

L'assignation de la variable `upperString` lui fait référencer la chaîne "Hello". L'invocation `Output(upperString)` passe à la fonction l'objet référencée par `upperString`, qui est notre vieille connaissance "Hello". La Figure 7.1 représente ce processus. À partir de là, l'effet est le même.

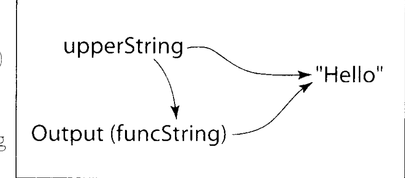


Figure 7.1 : L'invocation `Output (upperString)` copie la valeur de `upperString` dans `funcString`.

Passer plusieurs arguments à une fonction

Quand je demande à ma fille de laver la voiture, elle me donne en général plusieurs arguments. Comme elle passe beaucoup de temps sur le canapé pour y réfléchir, elle peut effectivement en avoir plusieurs à sa disposition.

Vous pouvez définir une fonction comportant plusieurs arguments de divers types. Considérez l'exemple suivant, `AverageAndDisplay()` :




```
// AverageAndDisplay
using System;
namespace Example
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // accède à la fonction membre
      AverageAndDisplay("UV 1", 3.5, "UV 2", 4.0);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // AverageAndDisplay - fait la moyenne de deux nombres associés
    //                     à leur nom et affiche le résultat
    public static void AverageAndDisplay(string s1, double d1,
                                         string s2, double d2)
```

```
    {
      double dAverage = (d1 + d2) / 2;
      Console.WriteLine("La moyenne de "  + s1
                      + " dont la valeur est " + d1
                      + " et de "           + s2
                      + " dont la valeur est " + d2
                      + " est égale à " + dAverage);
    }
  }
}
```

L'exécution de ce petit programme produit l'affichage suivant (le saut de ligne ne vient pas du programme) :

```
La moyenne de UV 1 dont la valeur est 3.5 et de UV 2
dont la valeur est 4 est égale à 3.75
Appuyez sur Entrée pour terminer...
```

La fonction `AverageAndDisplay()` est déclarée avec plusieurs arguments, dans l'ordre dans lequel ils doivent lui être passés.

Comme d'habitude, l'exécution de notre exemple de programme commence avec la première instruction qui suit `Main()`. La première ligne qui ne soit pas un commentaire dans `Main()` invoque la fonction `AverageAndDisplay()` en lui passant les deux chaînes "UV 1" et "UV 2", et les deux valeurs de type `double` 3.5 et 4.0.

La fonction `AverageAndDisplay()` calcule la moyenne des deux valeurs `double` `d1` et `d2`, qui lui ont été passées avec leurs noms respectifs contenus dans `s1` et `s2`, et cette moyenne est stockée dans `dAverage`.

Accorder la définition d'un argument et son utilisation

Dans un appel de fonction, l'ordre et le type des arguments doivent correspondre à la définition de la fonction. Ce qui suit est illicite et produit une erreur lors de la génération :




```
// AverageWithCompilerError -cette version ne se compile pas !
using System;
namespace AverageWithCompilerError
{
```

```
    public class Class1
    {
      public static void Main(string[] args)
      {
        // accède à la fonction membre
        AverageAndDisplay("UV 1", "UV 2", 3.5, 4.0);
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
      }
      // AverageAndDisplay - fait la moyenne de deux nombres associés
      //                     à leur nom et affiche le résultat
      public static void AverageAndDisplay(string s1, double d1,
                                           string s2, double d2)
      {
        double dAverage = (d1 + d2) / 2;
        Console.WriteLine("La moyenne de "  + s1
                        + " dont la valeur est " + d1
                        + " et de "          + s2
                        + " dont la valeur est " + d2
                        + " est égale à " + dAverage);
      }
    }
  }
```

C# ne peut pas faire correspondre les arguments qui sont passés dans l'appel à `AverageAndDisplay()` avec la définition de la fonction. La chaîne "UV 1" correspond bien au premier argument qui est de type `string` dans la définition de la fonction, mais pour le deuxième, la définition de la fonction demande un type `double` alors que c'est une chaîne qui est passée dans l'appel.

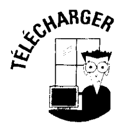
Il est facile de voir que j'ai simplement interverti le deuxième et le troisième argument dans l'appel de la fonction. Voilà ce que je n'aime pas avec ces ordinateurs : ils prennent littéralement tout ce que je leur dis. Je sais bien que je l'ai dit, mais ils pourraient comprendre ce que je voulais dire !

Surcharger une fonction ne signifie pas lui donner trop de travail



Vous pouvez donner le même nom à deux fonctions d'une même classe, *à condition* que leurs arguments soient différents. On appelle ça *surcharger* le nom de la fonction.

Voici un exemple de surcharge :




```
// AverageAndDisplayOverloaded - cette version montre que
//                               la fonction AverageAndDisplay
//                               peut être surchargée
using System;
namespace AverageAndDisplayOverloaded
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // accède à la première fonction membre
      AverageAndDisplay("mes points d'UV", 3.5, "vos points d'UV", 4.0);
      Console.WriteLine();
      // accède à la deuxième fonction membre
      AverageAndDisplay(3.5, 4.0);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // AverageAndDisplay - fait la moyenne de deux nombres associés
    //                     à leur nom et affiche le résultat
    public static void AverageAndDisplay(string s1, double d1,
                                         string s2, double d2)
    {
      double dAverage = (d1 + d2) / 2;
      Console.WriteLine("La moyenne de "  + s1
                      + " dont la valeur est " + d1);
      Console.WriteLine("et de "          + s2
                      + " dont la valeur est " + d2
                      + " est égale à " + dAverage);
    }
    public static void AverageAndDisplay(double d1, double d2)
    {
      double dAverage = (d1 + d2) / 2;
      Console.WriteLine("La moyenne de " + d1
                      + " et "          + d2
                      + " est égale à " + dAverage);
    }
  }
}
```

Ce programme définit deux versions de la fonction `AverageAndDisplay()`. Il invoque l'une puis l'autre en leur passant respectivement les arguments qu'il leur faut. C# peut identifier la fonction demandée par le programme en

comparant l'appel à la définition. Le programme se compile correctement, et son exécution donne l'affichage suivant :

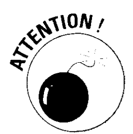
```
La moyenne de mes points d'UV dont la valeur est 3,5
et de vos points d'UV dont la valeur est 4 est égale à 3,75

La moyenne de 3,5 et 4 est égale à 3,75
Appuyez sur Entrée pour terminer...
```

En règle générale, C# ne permet pas à deux fonctions du même programme d'avoir le même nom. Après tout, comment pourrait-il deviner quelle fonction vous vouliez appeler ? Mais le nombre et le type des arguments de la fonction font partie de son nom. On pourrait appeler une fonction tout simplement `AverageAndDisplay()`, mais C# fait la différence entre les fonctions `AverageAndDisplay(string, double, string, double)` et `AverageAndDisplay(double, double)`. En voyant les choses de cette façon, il est clair que les deux fonctions sont différentes.

Implémenter des arguments par défaut

Bien souvent, vous voudrez pouvoir disposer de deux versions (ou plus) d'une même fonction. L'une pourra être la version compliquée qui offre une grande souplesse mais nécessite de nombreux arguments pour être appelée, dont plusieurs que l'utilisateur peut très bien ne même pas comprendre.



En pratique, quand on parle de "l'utilisateur" d'une fonction, il s'agit souvent du programmeur qui en fait usage. Ce n'est pas forcément le véritable utilisateur du programme.

Une autre version de la même fonction, bien qu'un peu fade, offrirait des performances acceptables, en remplaçant certains des arguments par des valeurs par défaut.

La surcharge d'un nom de fonction permet d'implémenter facilement des valeurs par défaut.

Examinez ces deux versions de la fonction `DisplayRoundedDecimal()` :

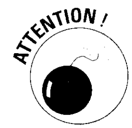
```
// FunctionsWithDefaultArguments - offre des variantes de la même
//                     fonction, certaines avec des arguments par
//                     défaut, en surchargeant le nom de la fonction
```

```
using System;
namespace FunctionsWithDefaultArguments
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // accède à la fonction membre
      Console.WriteLine("{0}", DisplayRoundedDecimal(12.345678M, 3));
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // DisplayRoundedDecimal - convertit en chaîne une valeur
    //                         décimale, avec le nombre spécifié
    //                         de chiffres significatifs
    public static string DisplayRoundedDecimal(decimal mValue,
                                      int nNumberOfSignificantDigits)
    {
      // commence pas arrondir le nombre sur la base du nombre
      // spécifié de chiffres significatifs
      decimal mRoundedValue =
                    decimal.Round(mValue,
                                  nNumberOfSignificantDigits);
      // Convertit en chaîne le résultat obtenu
      string s = Convert.ToString(mRoundedValue);
      return s;
    }
    public static string DisplayRoundedDecimal(decimal mValue)
    {
      // invoque DisplayRoundedDecimal(decimal, int),
      // en spécifiant le nombre de chiffres par défaut
      string s = DisplayRoundedDecimal(mValue, 2);
      return s;
    }
  }
}
```

La fonction `DisplayRoundedDecimal(decimal, int)` convertit en chaîne la valeur décimale qui lui est passée, avec le nombre spécifié de chiffres après la virgule. Comme les valeurs de type `decimal` sont très souvent utilisées pour représenter des valeurs monétaires, on utilise le plus souvent deux chiffres après la virgule. Aussi, la fonction `DisplayRoundedDecimal(decimal)` fait le même travail de conversion, mais en utilisant le paramètre par défaut de deux chiffres après la virgule, ce qui évite à l'utilisateur d'avoir seulement à se demander ce que veut dire le deuxième argument.



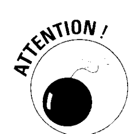
Remarquez que la version générique (`decimal`) de cette fonction fait son travail en appelant la version complète (`decimal, int`). La version générique trouve toute seule les arguments que le programmeur n'a pas envie de chercher dans la documentation.



Fournir des arguments par défaut présente plus d'intérêt que d'épargner simplement quelques efforts à un programmeur paresseux. Les recherches inutiles dans la documentation pour y trouver la signification d'arguments qui sont normalement définis par défaut distraient le programmeur de sa tâche principale et la lui rendent plus difficile, lui font perdre du temps, et augmentent le risque de produire des erreurs. Le programmeur qui est l'auteur d'une fonction comprend les relations entre ses arguments. C'est à lui que revient la charge d'en fournir une version simplifiée, plus facile à utiliser.

Passer des arguments d'un type valeur

Les types de variables de base, `int`, `double`, et `decimal`, sont appelés *types valeur*. Il y a deux manières de passer à une fonction des variables d'un type valeur. La forme par défaut consiste à *passer par valeur*. L'autre forme consiste à *passer par référence*.

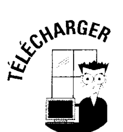


Les programmeurs ont leur manière de dire les choses. En parlant d'un type valeur, quand un programmeur dit "passer une variable à une fonction", cela signifie généralement "passer à une fonction la valeur d'une variable".

Passer par valeur des arguments d'un type valeur

Contrairement à une référence à un objet, une variable d'un type valeur comme `int` ou `double` est normalement passée par valeur, ce qui signifie que c'est la valeur contenue dans la variable qui est passée à la fonction, et non la variable elle-même.

Passer par valeur a pour effet que la modification de la valeur d'une variable d'un type valeur dans une fonction ne change pas à la valeur de cette variable dans le programme qui appelle la fonction.




```
// PassByValue - montre la sémantique du passage par valeur
using System;
namespace PassByValue
{
```

```
    public class Class1
    {
      // Update - essaie de modifier la valeur
      //          des arguments qui lui sont passés
      public static void Update(int i, double d)
      {
        i = 10;
        d = 20.0;
      }
      public static void Main(string[] args)
      {
        // déclare deux variables et les initialise
        int i = 1;
        double d = 2.0;
        Console.WriteLine("Avant l'appel à Update(int, double):");
        Console.WriteLine("i = " + i + ", d = " + d);
        // invoque la fonction
        Update(i, d);
        // remarquez que les valeurs 1 et 2.0 n'ont pas changé
        Console.WriteLine("Après l'appel à Update(int, double):");
        Console.WriteLine("i = " + i + ", d = " + d);
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
      }
    }
  }
```

L'exécution de ce programme produit l'affichage suivant :

```
Avant l'appel à Update(int, double):
i = 1, d = 2
Après l'appel à Update(int, double):
i = 1, d = 2
Appuyez sur Entrée pour terminer...
```

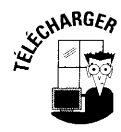
L'appel à `Update()` passe les valeurs 1 et 2.0, et non une référence aux variables `i` et `d`. Aussi, la modification de ces valeurs à l'intérieur de la fonction n'a aucun effet sur la valeur des variables dans la routine appelant la fonction.

Passer par référence des arguments d'un type valeur

Il est avantageux de passer par référence à une fonction un argument d'un type valeur, en particulier lorsque le programme appelant à besoin de donner à

cette fonction la possibilité de changer la valeur de la variable. Le programme qui suit, PassByReference, met en évidence cette possibilité.

C# donne au programmeur la possibilité de passer des arguments par référence en utilisant les mots-clés ref et out. C'est ce que montre une petite modification de l'exemple PassByValue de la section précédente.




```
// PassByReference - demonstrate pass by reference semantics
using System;
namespace PassByReference
{
  public class Class1
  {
    // Update - essaie de modifier la valeur
    //          des arguments qui lui sont passés
    public static void Update(ref int i, out double d)
    {
      i = 10;
      d = 20.0;
    }
    public static void Main(string[] args)
    {
      // déclare deux variables et les initialise
      int i = 1;
      double d;
      Console.WriteLine("Avant l'appel à Update(ref int, out double):");
      Console.WriteLine("i = " + i + ", d n'est pas initialisé");
      // invoque la fonction
      Update(ref i, out d);
      // remarquez que les valeurs 1 et 2.0 n'ont pas changé
      Console.WriteLine("Après l'appel à Update(ref int, out double):");
      Console.WriteLine("i = " + i + ", d = " + d);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

Le mot-clé ref indique que c'est une référence que C# doit passer à i, et non la valeur contenue dans cette variable. Il en résulte que les modifications apportées à cette valeur dans la fonction sont exportées dans le programme qui l'appelle.

De façon similaire, le mot-clé out dit "restituer par référence, mais peu m'importe quelle était la valeur initiale, puisque de toute façon je vais la remplacer" (ça fait

beaucoup de choses en trois mots). Le mot-clé `out` est applicable lorsque la fonction ne fait que retourner une valeur au programme appelant.

L'exécution de ce programme produit l'affichage suivant :

```
Avant l'appel à Update(ref int, out double):
i = 1, d n'est pas initialisé
Après l'appel à Update(ref int, out double):
i = 10, d = 20
Appuyez sur Entrée pour terminer...
```



Un argument `out` est toujours `ref`.

Notez que les valeurs initiales de `i` et de `d` sont écrasées dans la fonction `Update()`. De retour dans `Main()`, ces variables ont reçu leur valeur modifiée. Comparez cela à la fonction `PassByValue()`, dans laquelle les variables ne reçoivent pas leur valeur modifiée.

Ne passez pas une variable par référence à une fonction deux fois en même temps

Jamais, sous aucun prétexte, vous ne devez passer deux fois par référence la même variable dans un même appel à une fonction. La raison en est plus difficile à expliquer qu'à montrer. Examinez la fonction `Update()` suivante :




```
// PassByReferenceError - montre une situation d'erreur potentielle
//                        quand on appelle une fonction avec
//                        des arguments passés par référence
using System;
namespace PassByReferenceError
{
  public class Class1
  {
    // Update - essaie de modifier la valeur
    //          des arguments qui lui sont passés
    public static void DisplayAndUpdate(ref int nVar1, ref int nVar2)
    {
      Console.WriteLine("La valeur initiale de nVar1 est " + nVar1);
      nVar1 = 10;
      Console.WriteLine("La valeur initiale de nVar2 est " + nVar2);
      nVar2 = 20;
    }
    public static void Main(string[] args)
    {
      // déclare deux variables et les initialise
```

```
            int n = 1;
            Console.WriteLine("Avant l'appel à Update(ref n, ref n):");
            Console.WriteLine("n = " + n);
            Console.WriteLine();
            // invoque la fonction
            DisplayAndUpdate(ref n, ref n);
            // remarquez que les valeurs 1 et 2.0 n'ont pas changé
            Console.WriteLine();
            Console.WriteLine("Après l'appel à Update(ref n, ref n):");
            Console.WriteLine("n = " + n);
            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
        }
    }
}
```

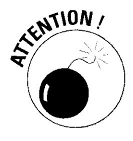
Update(ref int, ref int) est maintenant déclarée pour accepter par référence deux arguments de type int, ce qui, en soi, n'est pas un problème. Le problème se produit lorsque la fonction Main() invoque Update() en lui passant la même variable pour les deux arguments. Dans la fonction, Update() modifie nVar1, ce qui, par référence à n, remplace sa valeur initiale de 1 par la valeur 10. En même temps, Update() modifie nVar2, alors que la valeur de n, à laquelle elle se réfère, a déjà été modifiée en recevant la valeur 10.

Ce qui est mis en évidence dans l'exemple suivant :

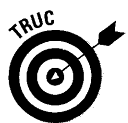
```
Avant l'appel à Update(ref n, ref n):
n = 1

La valeur initiale de nVar1 est 1
La valeur initiale de nVar2 est 10

Après l'appel à Update(ref n, ref n):
n = 20
Appuyez sur Entrée pour terminer...
```



Ce qui se passe exactement au cours de ce petit jeu de scène entre n, nVar1, et nVar2, est à peu près aussi évident que la danse nuptiale d'un oiseau exotique. Ni l'auteur de la fonction Update(), ni le programmeur qui l'utilise, n'ont prévu ce curieux résultat. Autrement dit, ne le faites pas.



Il n'y a aucun inconvénient à passer une même valeur à plusieurs arguments différents dans un même appel de fonction si toutes les variables sont passées par valeur.

Pourquoi y a-t-il certains arguments qui sortent mais n'entrent pas ?

C# veille sans relâche à empêcher le programmeur de faire une bêtise. L'une des bêtises que peut faire un programmeur est d'oublier d'initialiser une variable avant de commencer à s'en servir (c'est particulièrement vrai pour les variables utilisées comme compteur). Lorsque vous essayez d'utiliser une variable que vous avez déclarée mais pas initialisée, C# génère une erreur.

```
int nVariable;
Console.WriteLine("ceci est une erreur " + nVariable);
nVariable = 1;
Console.WriteLine("mais ceci n'en est pas une " + nVariable);
```

Mais C# ne peut pas assurer la surveillance des variables à l'intérieur d'une fonction :

```
void SomeFunction(ref int nVariable)
{
  Console.WriteLine("ceci est-il une erreur ? " + nVariable);
}
```

Comment `SomeFunction()` pourrait-elle savoir si `nVariable` a été initialisée avant d'être passée dans l'appel de la fonction ? Elle ne le peut pas. Au lieu de cela, C# examine la variable dans l'appel. Par exemple, l'appel suivant génère une erreur de compilation :

```
int nUninitializedVariable;
SomeFunction(ref nUninitializedVariable);
```

Si C# avait accepté cet appel, `SomeFunction()` se serait vu passer une référence à une variable non initialisée (donc n'ayant aucun sens). Le mot-clé `out` permet aux deux parties de se mettre d'accord sur le fait qu'aucune valeur n'a encore été assignée à la variable. L'exemple suivant se compile sans problème :

```
int nUninitializedVariable;
SomeFunction(out nUninitializedVariable);
```

Au passage, il est licite de passer en tant qu'argument `out` une variable initialisée :

```
int nInitializedVariable = 1;
SomeFunction(out nInitializedVariable);
```

La valeur de `nInitializedVariable` sera écrasée dans `SomeFunction()`, mais il n'y a aucun risque que soit passée une valeur dépourvue de sens.

Retourner une valeur à l'expéditeur

Dans bien des opérations du monde réel, des valeurs sont créées pour être retournées à celui qui les a envoyées. Par exemple, `sin()` accepte un argument pour lequel elle calcule la fonction trigonométrique sinus, et retourne la valeur correspondante. Une fonction dispose de deux moyens pour retourner une valeur à celui qui l'a appelée. La plus courante est l'utilisation du champ `return`, mais il y a une autre méthode qui utilise l'*appel par référence*.

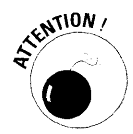
Utiliser `return` pour retourner une valeur

L'exemple suivant montre une petite fonction qui retourne la moyenne des arguments qui lui sont passés :

```
public class Example
{
  public static double Average(double d1, double d2)
  {
    double dAverage = (d1 + d2) / 2;
    return dAverage;
  }
  public static void Test()
  {
    double v1 = 1.0;
    double v2 = 3.0;
    double dAverageValue = Average(v1, v2);
    Console.WriteLine("La moyenne de " + v1
                + " et de " + v2 + " est "
                + dAverageValue);
    // ceci fonctionne également
    Console.WriteLine("La moyenne de " + v1
                + " et de " + v2 " + est "
                + Average(v1, v2));
  }
}
```

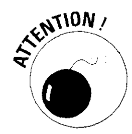
Remarquez tout d'abord que je déclare cette fonction comme `public double Average()`. Le `double` qui précède le nom signifie que la fonction `Average()` retourne une valeur en double précision à celui qui l'a appelée.

La fonction `Average()` attribue les noms `d1` et `d2` aux valeurs en double précision qui lui sont passées. Elle crée une variable `dAverage` à laquelle elle assigne la moyenne de `d1` et `d2`, puis elle retourne au programme appelant la valeur contenue dans `dAverage`.



Dans ce cas, certains diraient que "la fonction retourne `dAverage`". C'est un abus de langage, mais un raccourci d'usage courant. Dire que `dAverage` ou tout autre variable est passée ou retournée où que ce soit n'a aucun sens. Dans ce cas, c'est la valeur contenue dans `dAverage` qui est retournée au programme appelant.

L'appel à `Average()` dans la fonction `Test()` semble identique à n'importe quel autre appel de fonction, mais la valeur de type `double` retournée par `Average()` est stockée dans la variable `dAverageValue`.



Une fonction qui retourne une valeur, comme `Average()`, ne peut pas la retourner en rencontrant la dernière parenthèse fermante de la fonction. Si c'était le cas, comment ferait C# pour savoir quelle valeur retourner ?

Retourner une valeur en utilisant un passage par référence

Une fonction peut aussi retourner une ou plusieurs valeurs à la routine qui l'appelle en utilisant les mots-clés `ref` et `out`. Regardez l'exemple `Update()` décrit dans la section "Passer par référence des arguments d'un type valeur", plus haut dans ce chapitre :

```
// Update - essaie de modifier la valeur
//          des arguments qui lui sont passés
public static void Update(ref int i, out double d)
{
  i = 10;
  d = 20.0;
}
```

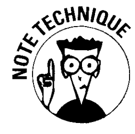
La fonction est déclarée `void`, comme si elle ne retournait pas de valeur au programme appelant, mais puisque la variable `i` est déclarée `ref` et la variable `d` est déclarée `out`, toute modification apportée à ces deux variables dans `Update()` est conservée dans le programme appelant.

Quand utiliser `return` *et quand utiliser* `out` *?*

Vous pensez peut-être : "Une fonction peut retourner une valeur au programme appelant, ou bien utiliser pour cela `out` ou `ref`. Quand faut-il utiliser `return`, et quand faut-il utiliser `out` ?" Après tout, j'aurais très bien pu écrire de la façon suivante la fonction `Average()` :

```
public class Example
{
  public static void Average(out double dResults, double d1, double d2)
  {
    dResults = (d1 + d2) / 2;
  }
  public static void Test()
  {
    double v1 = 1.0;
    double v2 = 3.0;
    double dAverageValue;
    Average(dAverageValue, v1, v2);
    Console.WriteLine("La moyenne de " + v1
                    + " et de " + v2 + " est "
                    + dAverageValue;
  }
}
```

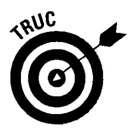
C'est le plus souvent par l'instruction `return` que vous allez retourner une valeur au programme appelant, plutôt que par la directive `out`, bien qu'il soit difficile de contester que ça revient au même.



Utiliser `out` avec une variable d'un type valeur comme `double` nécessite un procédé supplémentaire que l'on appelle *boxing*, dont la description sort du cadre de ce livre. Toutefois, l'efficacité ne doit pas être un facteur clé dans votre choix.

C'est typiquement quand une fonction retourne plusieurs valeurs au programme appelant que vous allez utiliser `out`. Par exemple :

```
public class Example
{
  public static void AverageAndProduct(out double dAverage,
                                       out double dProduct,
                                       double d1, double d2)
  {
    dAverage = (d1 + d2) / 2;
    dProduct = d1 * d2;
  }
}
```



Une fonction qui retourne à elle seule plusieurs valeurs est une créature que l'on ne rencontre pas aussi souvent qu'on pourrait le croire. Une telle fonction est souvent encapsulée dans un objet de classe ou dans un tableau de valeurs.

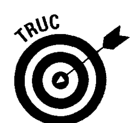
Définir une fonction qui ne retourne pas de valeur

La déclaration `public double Average(double, double)` déclare une fonction `Average()`, qui retourne la moyenne de ses arguments sous forme `double`.

Il y a des fonctions qui ne retournent aucune valeur au programme appelant. Une fonction que nous avons utilisée plus haut comme exemple, `AverageAndDisplay()`, affiche la moyenne des arguments qui lui sont passés, mais ne retourne pas cette moyenne au programme appelant. Ce n'est peut-être pas une bonne idée, mais telle n'est pas ici la question. Au lieu de laisser en blanc le type retourné, une fonction comme `AverageAndDisplay()` est déclarée de la façon suivante :

```
public void AverageAndDisplay(double, double)
```

Le mot-clé `void`, placé à l'endroit où apparaîtrait normalement le type retourné, signifie *pas de type*. Autrement dit, la déclaration `void` indique que la fonction `AverageAndDisplay()` ne retourne aucune valeur au programme appelant.



Une fonction qui ne retourne aucune valeur est appelée une *fonction sans type* (*void function*). Par opposition, une fonction qui retourne une valeur est appelée une *fonction typée* (*non-void function*).

Une fonction typée doit restituer le contrôle au programme appelant par une instruction `return` suivie par la valeur à retourner. Une fonction sans type n'a aucune valeur à retourner. Elle restitue le contrôle lorsqu'elle rencontre un `return` qui n'est suivi d'aucune valeur. Par défaut, une fonction sans type se termine automatiquement (restitue le contrôle au programme appelant) lorsque le contrôle atteint l'accolade fermante qui en indique la fin.

Examinez la fonction `DisplayRatio()` :

```
public class Example
{
  public static void DisplayRatio(double dNumerator,
                                  double dDenominator)
  {
    // si le dénominateur est égal à zéro...
    if (dDenominator == 0.0)
    {
```

```
        // . . . affiche un message d'erreur et...
        Console.WriteLine(
                    "Le dénominateur d'un quotient ne peut pas être 0");
        // . . . retourne à la fonction appelante
        return;
      }
      // ceci n'est exécuté que si dDenominator n'est pas nul
      double dRatio = dNumerator / dDenominator;
      Console.WriteLine("Le quotient " + dNumerator
                      + " sur " + dDenominator
                      + " est égal à " + dRatio);
    }
  }
```

La fonction `DisplayRatio()` regarde si la valeur de `dDenominator` est égale à zéro. Si c'est le cas, elle affiche un message d'erreur et restitue le contrôle au programme appelant, sans essayer de calculer le ratio. Sans cette précaution, la valeur du numérateur serait divisée par zéro, et produirait une erreur du processeur, que l'on appelle aussi du nom plus imagé de *processor upchuck*. (Autrement dit, "le processeur dégueule". Désolé.)

Si `dDenominator` n'est pas égal à zéro, la fonction affiche le ratio. La parenthèse fermante qui suit immédiatement l'instruction `WriteLine()` est celle qui indique la fin de la fonction `DisplayRatio()`, donc joue le rôle d'une instruction `return`.

Référence à `null` et référence à zéro

Lorsqu'elle est créée, une variable de référence se voit assigner la valeur par défaut `null`. Mais une référence à `null` n'est pas la même chose qu'une référence à zéro. Par exemple, les deux références ci-dessous sont complètement différentes :

```
class Example
{
  int nValue;
}

// crée une référence null ref1
Example ref1;

// crée maintenant une référence à un objet de valeur nulle
Example ref2 = new Example();
ref2.nValue = 0;
```

La variable `ref1` est à peu près aussi vide que mon portefeuille. Elle pointe vers l'objet `null`, c'est-à-dire vers aucun objet. En revanche, `ref2` pointe vers un objet dont la valeur est 0.

Cette différence est beaucoup moins claire dans l'exemple suivant :

```
string s1;
string s2 = "";
```

C'est essentiellement la même chose : `s1` pointe vers l'objet `null`, et `s2` pointe vers une chaîne vide. La différence est significative, comme le montre la fonction suivante :

```
// Test - modules de test pour utiliser la bibliothèque TestLibrary
namespace Test
{
  using System;

  public class Class1
  {
    public static int Main(string[] strings)
    {
      Console.WriteLine("Ce programme utilise " +
                        "la fonction TestString()");
      Console.WriteLine();
      Example exampleObject = new Example();

      Console.WriteLine("Passage d'un objet null :");
      string s = null;
      exampleObject.TestString(s);
      Console.WriteLine();

      // passe maintenant à la fonction une chaîne vide
      Console.WriteLine("Passage d'une chaîne vide :");
      exampleObject.TestString("");
      Console.WriteLine();

      // enfin, passage d'une véritable chaîne
      Console.WriteLine("Passage d'une véritable chaîne :");
      exampleObject.TestString("chaîne de test");
      Console.WriteLine();

      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
```

```
    class Example
    {
      public void TestString(string sTest)
      {
        // commence par vérifier si la chaîne est vide
        if (sTest == null)
        {
          Console.WriteLine("sTest est vide");
          return;
        }

        // vérifie si sTest pointe vers une chaîne vide
        if (String.Compare(sTest, "") == 0)
        {
          Console.WriteLine("sTest référence une chaîne vide");
          return;
        }

        // puisque tout va bien, affiche la chaîne
        Console.WriteLine("sTest se réfère à : '" + sTest + "'");
      }
    }
  }
```

La fonction `TestString()` utilise la comparaison `sTest == null` pour savoir si une chaîne a pour valeur `null`. Mais `TestString()` doit utiliser la fonction `Compare()` pour tester si une chaîne vide (`Compare()` retourne un 0 si les deux chaînes qui lui sont passées sont égales).

Ce programme affiche les résultats suivants :

```
Ce programme utilise la fonction TestString()

Passage d'un objet null :
sTest est vide

Passage d'une chaîne vide :
sTest référence une chaîne vide

Passage d'une véritable chaîne :
se réfère à : 'test string'

Appuyez sur Entrée pour terminer...
```

La question de Main() *: passer des arguments à un programme*

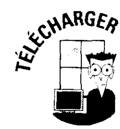
Examinez toutes les applications console de ce livre. L'exécution commence toujours par Main(). Sa déclaration vous dit clairement de quoi il s'agit :

```
public static void Main(string[] args)
{
  //  . . . emplacement de votre programme. . .
}
```

Main() est une fonction statique ou une fonction de classe de la classe Class1, définie par l'Assistant Applications de Visual Studio. Main() ne retourne aucune valeur et accepte comme arguments un tableau d'objets de type string. Que sont ces chaînes ?

Pour exécuter une application console, l'utilisateur entre le nom du programme. Après ce nom, il a la possibilité d'ajouter des arguments. C'est ce que vous voyez tout le temps, par exemple avec une commande comme copy monfichier C:\mondossier, qui copie le fichier monfichier dans le dossier mondossier du répertoire racine du lecteur C.

Comme vous pouvez le voir dans l'exemple DisplayArguments suivant, le tableau de valeurs de type string passé à Main() constitue les arguments du programme :




```
// DisplayArguments - affiche les arguments qui sont passés
//                    au programme
using System;
namespace DisplayArguments
{
  public class Test
  {
    public static int Main(string[] args)
    {
      // compte le nombre d'arguments
      Console.WriteLine("Ce programme a {0} arguments",
                        args.Length);
      // les arguments sont :
      int nCount = 0;
      foreach(string arg in args)
      {
```

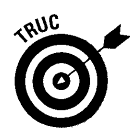
```
                Console.WriteLine("L'argument {0} est {1}",
                                  nCount++, arg);
            }
            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
            return 0;
        }
    }
}
```

Ce programme commence par afficher la longueur du tableau args. Cette valeur correspond au nombre des arguments passés à la fonction. Le programme effectue alors une boucle sur tous les éléments de args, affichant successivement chacun d'entre eux sur la console.

L'exécution de ce programme peut produire par exemple les résultats suivants :

```
DisplayArguments /c arg1 arg2
Ce programme a 3 arguments
L'argument 0 est /c
L'argument 1 est arg1
L'argument 2 est arg2
Appuyez sur Entrée pour terminer...
```

Vous pouvez voir que le nom du programme lui-même n'apparaît pas dans la liste des arguments (il existe aussi une fonction qui permet au programme de trouver dynamiquement son propre nom). D'autre part, l'option /c n'est pas traitée différemment des autres arguments. C'est le programme lui-même qui se charge de l'analyse des arguments qui lui sont passés.



La plupart des applications console autorisent l'utilisation d'options qui permettent de contrôler certains détails du fonctionnement du programme.

Passer des arguments à l'invite de DOS

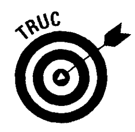
Pour exécuter à partir de l'invite de DOS le programme DisplayArguments, suivez ces étapes :

1. **Cliquez sur Démarrer/Programmes/Accessoires/Invite de commandes.**

Vous devez voir apparaître une fenêtre à fond noir contenant la respectable antiquité `C:\>` , suivie d'un curseur clignotant.

2. **Naviguez jusqu'au dossier contenant le projet** `DisplayArguments` **en tapant au clavier** \Programmes C#\DisplayArguments. **(Le dossier par défaut pour les exemples de ce programme est** `Programmes C#`**. Utilisez le vôtre si vous en avez choisi un autre.)**

L'invite devient `C:\C#Programs\DisplayArguments>`.



Si vous ne le trouvez pas, utilisez Windows pour rechercher le programme. Dans l'Explorateur Windows, cliquez du bouton droit sur le dossier racine C:\, et sélectionnez Rechercher comme le montre la Figure 7.2.

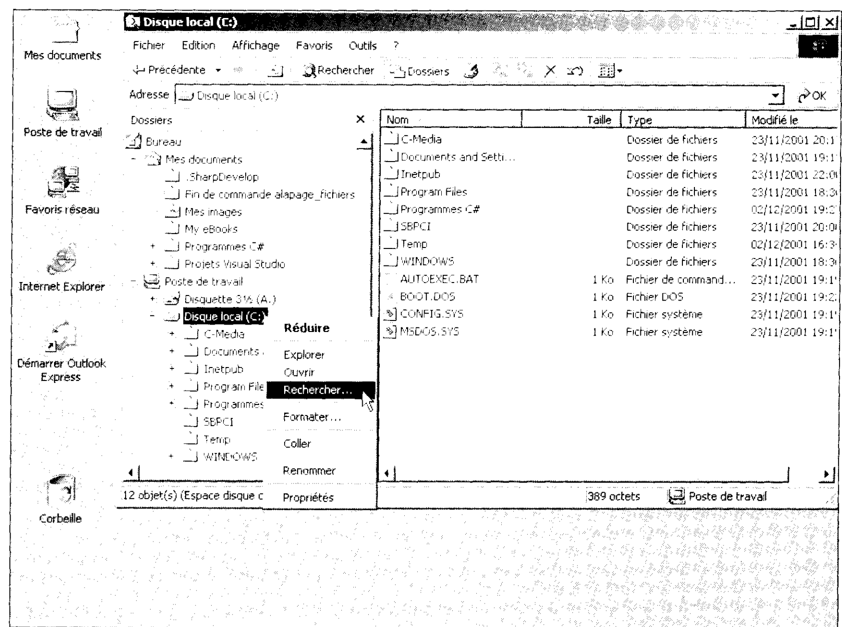


Figure 7.2 : L'utilitaire de recherche de Windows est une aide précieuse pour retrouver ses fichiers.

Dans la boîte de dialogue qui apparaît, Entrée **DisplayArguments.exe**, et cliquez sur `Rechercher`. Le nom du fichier apparaît en haut du volet de droite de la fenêtre Résultats de la recherche, comme le montre la Figure 7.3. Ignorez le fichier `DisplayArguments.exe` qui se trouve dans le dossier `obj`. Vous aurez peut-être besoin de faire défiler horizontalement le contenu de la fenêtre pour voir le chemin d'accès complet au fichier s'il est enfoui profondément dans l'arborescence des dossiers. C'est souvent le cas si vous stockez vos fichiers dans le dossier Mes documents.

Visual Studio .NET place normalement les exécutables qu'il génère dans un sous-dossier `bin\Debug`. Mais si vous modifiez la configuration, ce dossier peut tout aussi bien être `bin\release` ou un autre.

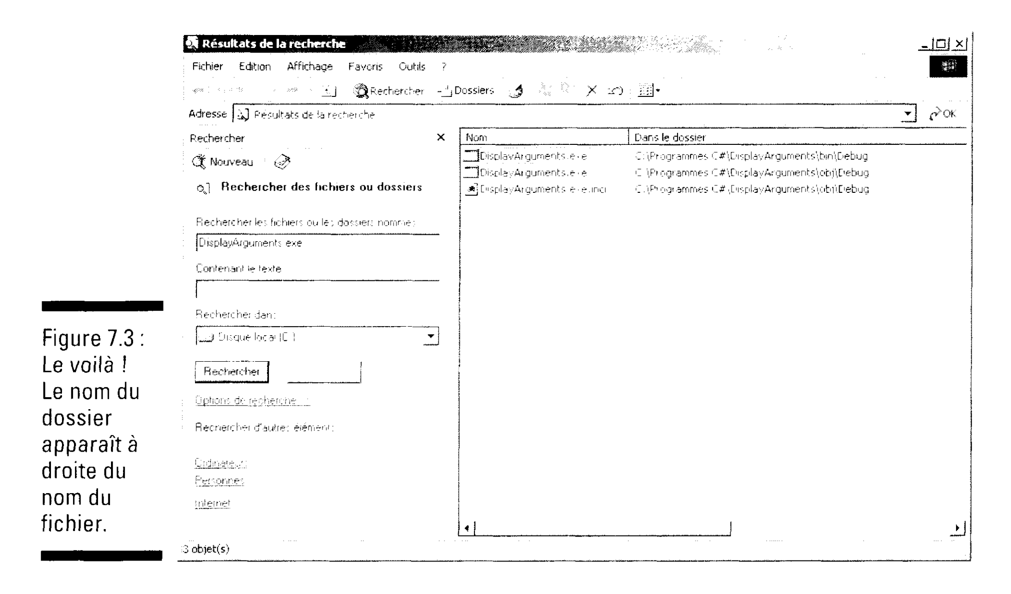
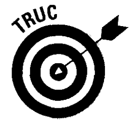


Figure 7.3 : Le voilà ! Le nom du dossier apparaît à droite du nom du fichier.

3. **Dans la fenêtre d'invite de commandes, tapez** cd debug\bin **pour passer dans le répertoire qui contient les exécutables.**

   L'invite devient `C:\C#Programs\DisplayArguments\bin\Debug`.



Windows accepte sans problème les noms de fichier ou de répertoire contenant des espaces, mais DOS peut avoir du mal à s'y retrouver. Si vous avez un nom de fichier ou de répertoire contenant des espaces, vous devez l'entourer par des guillemets. Par exemple, pour naviguer jusqu'à un fichier qui se trouve dans le dossier Mes fichiers, j'utiliserai une commande comme celle-ci :

```
cd \"Mes fichiers"
```

4. **À l'invite de commandes, tapez** DisplayArguments /c arg1 arg2 **pour exécuter le programme DisplayArguments.exe.**

   Le programme doit répondre en affichant les résultats montrés par la Figure 7.4.

```
Invite de commandes - DisplayArguments /c arg1 arg2
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd Programmes C#\DisplayArguments\bin\Debug\

C:\Programmes C#\DisplayArguments\bin\Debug>DisplayArguments /c arg1 arg2
Ce programme a 3 arguments
L'argument 0 est /c
L'argument 1 est arg1
L'argument 2 est arg2
Appuyez sur Entrée pour terminer...
```

Figure 7.4 : L'exécution de `DisplayArguments` à partir de l'invite du DOS affiche les arguments que vous avez passés au programme.

Passer des arguments à partir d'une fenêtre

Vous pouvez exécuter un programme comme `DisplayArguments` en tapant son nom dans la ligne de commande d'une fenêtre de commandes. Vous pouvez aussi l'exécuter à partir de l'interface Windows en double-cliquant sur le nom du fichier du programme, dans une fenêtre ou dans l'Explorateur Windows.

Comme le montre la Figure 7.5, un double clic sur le fichier `DisplayArguments` exécute le programme comme si vous aviez entré son nom sans arguments sur la ligne de commande :

```
Ce programme a 0 argument
Appuyez sur Entrée pour terminer...
```

Pour terminer le programme et fermer la fenêtre, appuyez sur Entrée.

Faire glisser et déposer un ou plusieurs fichiers sur le fichier `DisplayArguments.exe` exécute le programme comme si vous aviez entré **DisplayArguments *noms des fichiers*** sur la ligne de commande. (Pour faire glisser et déposer plusieurs fichiers à la fois, commencez par les sélectionner en cliquant successivement sur chacun d'eux tout en maintenant enfoncée la touche Ctrl, comme le montre la Figure 7.6 ; puis faites glisser l'ensemble pour le déposer sur `DisplayArguments.exe`.) Faire glisser et déposer simultanément les fichiers `arg1.txt` et `arg2.txt` sur `DisplayArguments.exe` provoque l'exécution du programme avec plusieurs arguments comme le montre la Figure 7.7.

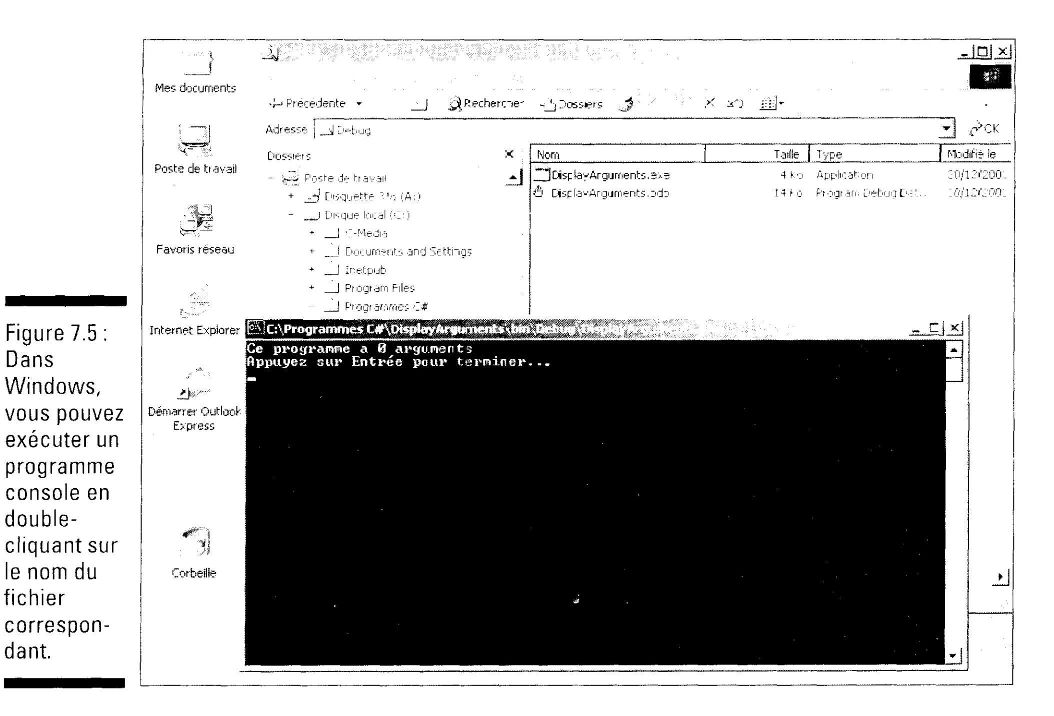


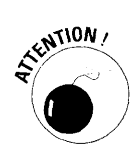
Figure 7.5 : Dans Windows, vous pouvez exécuter un programme console en double-cliquant sur le nom du fichier correspondant.

Figure 7.6 : Dans Windows, vous pouvez faire glisser et déposer un fichier sur un programme console pour déclencher son exécution.

La Figure 7.7 montre le résultat de l'exécution de `DisplayArguments` en déposant dessus les fichiers `arg1.txt` et `arg2.txt`.



Figure 7.7 : Faire glisser des fichiers sur le nom d'un programme produit le même résultat que si vous l'aviez exécuté à partir de la ligne de commande en lui passant les noms des fichiers correspondants.



Remarquez que Windows passe les fichiers à `DisplayArguments` dans un ordre quelconque.

Passer des arguments à partir de Visual Studio .NET

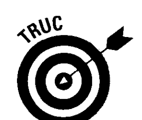
Pour exécuter un programme à partir de Visual Studio .NET, commencez par vous assurer que le programme est généré sans erreurs. Sélectionnez Générer/Générer, et regardez si la fenêtre Sortie affiche des erreurs. La réponse satisfaisante est `Génération : 1 a réussi, 0 a échoué, 0 a été ignoré`. Si ce n'est pas ça, votre programme ne démarrera pas.

À partir de là, l'exécution de votre programme sans lui passer d'arguments est un jeu d'enfant. Si la génération a réussi, sélectionnez Déboguer/Démarrer (ou appuyez sur F5), et le programme démarre.

Par défaut, c'est sans arguments que Visual Studio exécute un programme. Si ce n'est pas ce que vous voulez, vous devez indiquer à Visual Studio les arguments à utiliser :

1. **Ouvrez l'Explorateur de solutions en sélectionnant Affichage/ Explorateur de solutions.**

   La fenêtre de l'Explorateur de solutions affiche une description de votre *solution*. Une solution se compose d'un ou plusieurs projets. Chaque projet est la description d'un programme. Par exemple, le projet `DisplayArguments` dit que `Class1.cs` est l'un des fichiers de ce programme et que celui-ci est une application console. Un projet contient aussi d'autres propriétés, dont les arguments à utiliser pour exécuter le programme avec Visual Studio.



   C'est au Chapitre 17 que je décris le fichier de solution.

2. **Cliquez du bouton droit sur `DisplayArguments`, et sélectionnez Propriétés dans le menu qui apparaît, comme le montre la Figure 7.8.**

   Une fenêtre comme celle de la Figure 7.9 apparaît, montrant beaucoup de propriétés du projet avec lesquelles vous pouvez jouer. S'il vous plaît, ne le faites pas.

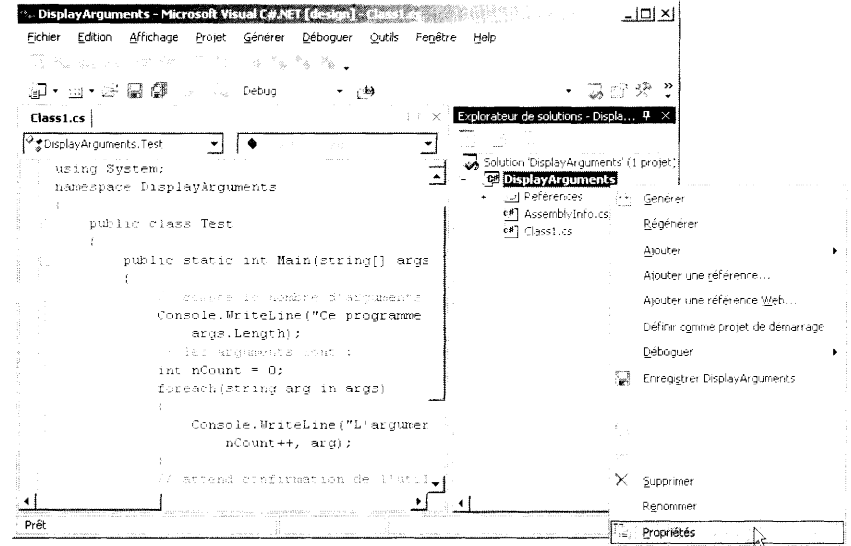


Figure 7.8 : Pour accéder aux propriétés d'un projet, cliquez du bouton droit sur son nom.

3. **Dans le volet de gauche de la fenêtre Pages de propriétés, sous le dossier Propriétés de configuration, sélectionnez Débogage.**

Dans le volet de droite, dans la rubrique Options de démarrage, il y a un champ nommé `Arguments de la ligne de commande`.

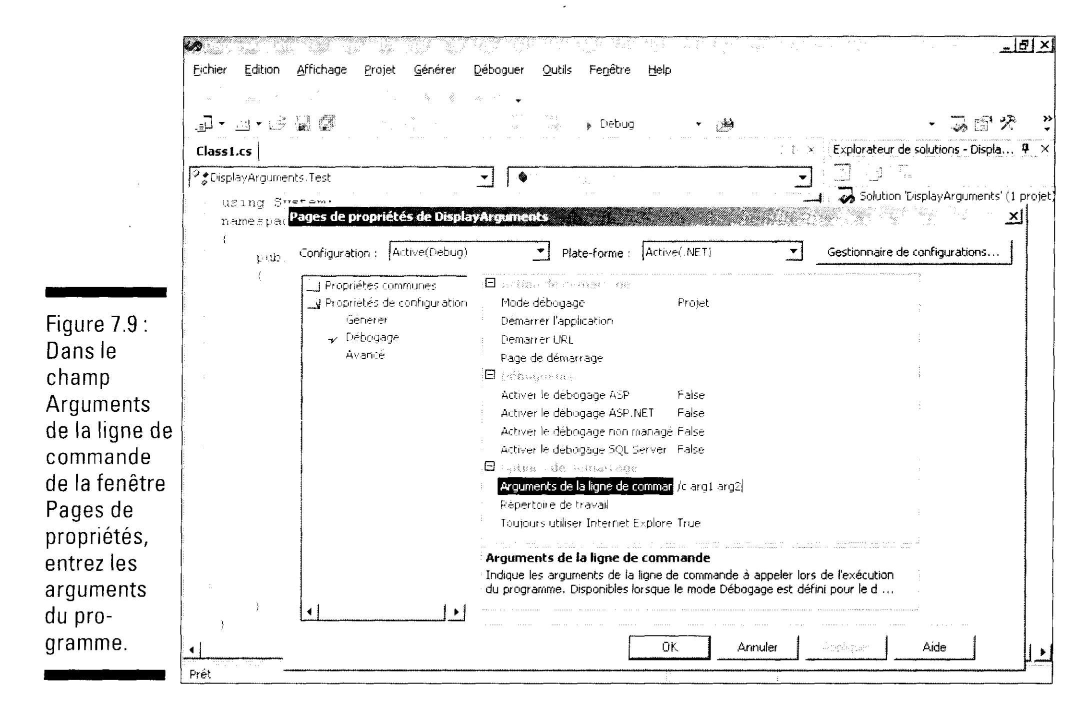


Figure 7.9 : Dans le champ Arguments de la ligne de commande de la fenêtre Pages de propriétés, entrez les arguments du programme.

4. **Tapez les arguments que vous voulez passer à votre programme quand il est lancé par Visual Studio.**

   Dans la Figure 7.9, ce sont les arguments `/c arg1 arg2` qui seront passés au programme.

5. **Cliquez sur OK, puis exécutez le programme normalement en sélectionnant Déboguer/Démarrer.**

   Comme le montre la Figure 7.10, Visual Studio ouvre une fenêtre DOS avec les résultats attendus :

```
Ce programme a 3 arguments
L'argument 0 est /c
L'argument 1 est arg1
L'argument 2 est arg2
Appuyez sur Entrée pour terminer...
```

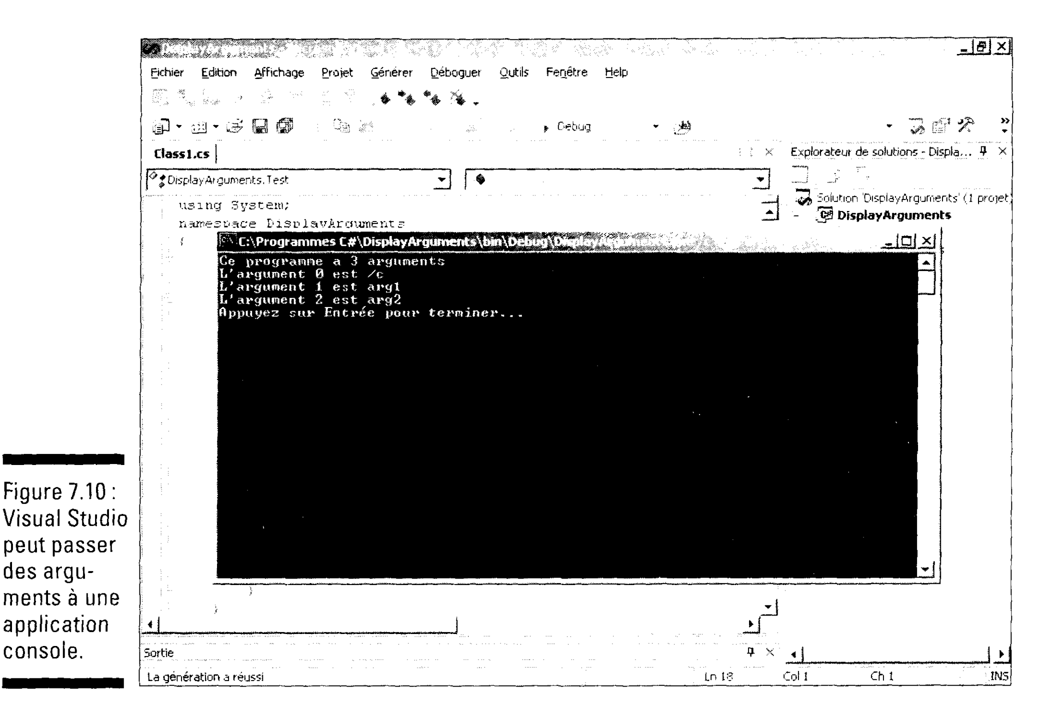


Figure 7.10 : Visual Studio peut passer des arguments à une application console.

La seule différence entre la sortie de l'exécution d'un programme à partir de Visual Studio .NET et la sortie de l'exécution du même programme à partir de la ligne de commande est l'absence du nom du programme dans l'affichage.

La fonction `WriteLine()`

Vous avez peut-être remarqué que l'instruction `WriteLine()` que vous avez utilisée jusqu'à maintenant dans divers programmes n'est rien d'autre qu'un appel à une fonction, invoquée avec ce que l'on appelle une classe `Console` :

```
Console.WriteLine("ceci est un appel de fonction");
```

`WriteLine()` est l'une des nombreuses fonctions prédéfinies offertes par l'environnement C#. `Console` est une classe prédéfinie qui se réfère à la console de l'application.

L'argument de la fonction `WriteLine()` que vous avez utilisée jusqu'ici dans divers exemples est une simple chaîne. L'opérateur "+" permet aux programmeurs de combiner des chaînes, ou de combiner une chaîne et une variable intrinsèque avant que le résultat de cette opération soit passé à `WriteLine()` :

```
string s = "Sarah"
Console.WriteLine("Je m'appelle " + s + " et j'ai " + 3 + "ans.");
```

Tout ce que voit `WriteLine()` dans cet exemple est "Je m'appelle Sarah et j'ai 3 ans."

Dans une autre forme, `WriteLine()` offre un ensemble d'arguments plus souple :

```
Console.WriteLine("Je m'appelle {0} et j'ai {1} ans.",
                  "Sarah", 3);
```

Ici, la chaîne "Sarah" est insérée à l'endroit où apparaît le symbole {0}. Le zéro se réfère au premier argument qui suit la chaîne elle-même. Le nombre entier 3 est inséré à l'endroit marqué par {1}. Cette forme est plus efficace que l'exemple précédent, car la concaténation de chaînes n'est pas une chose aussi facile qu'il y paraît. C'est une tâche qui prend du temps, mais il faut bien que quelqu'un le fasse.

Il n'y aurait pas grand-chose d'autre à en dire si c'était là la seule différence. Mais cette deuxième forme de `WriteLine()` offre également différentes possibilités de contrôle du format de sortie. Je les décrirai au Chapitre 9.

Chapitre 8

Méthodes de classe

Dans ce chapitre :

Passer un objet à une fonction.

Convertir en méthode une fonction membre.

Qu'est-ce que `this` ?

Créer une très belle documentation.

Les fonctions décrites au Chapitre 7 sont un excellent moyen de diviser un problème de programmation en éléments plus petits et plus maîtrisables. La possibilité de passer à une fonction et de récupérer des valeurs entières ou en virgule flottante permet au code de l'application de communiquer avec elle.

Quelques variables ne peuvent communiquer que les informations qu'elles contiennent. Un programme orienté objet repose sur le regroupement d'informations dans des objets. C'est pour cette raison que C# offre un moyen pratique et élégant de communiquer à des fonctions des objets de classe.

Passer un objet à une fonction

Vous pouvez passer une référence à un objet comme argument à une fonction de la même manière qu'une variable d'un type valeur, à une différence près : un objet est toujours passé par référence.

Le petit programme suivant montre comment passer un objet à une fonction :




```
// PassObject - montre comment passer un objet
//              à une fonction
using System;
namespace PassObject
{
  public class Student
  {
    public string sName;
  }
  public class Class1
  {
    public static void Main(string[] args)
    {
      Student student = new Student();
      // définit le nom en y accédant directement
      Console.WriteLine("La première fois :");
      student.sName = "Madeleine";
      OutputName(student);
      // change le nom en utilisant une fonction
      Console.WriteLine("Après avoir été modifié :");
      SetName(student, "Willa");
      OutputName(student);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // OutputName - affiche le nom de l'étudiant
    public static void OutputName(Student student)
    {
      //affiche le nom de l'étudiant courant
      Console.WriteLine("Le nom de l'étudiant est {0}", student.sName);
    }
    // SetName - modifie le nom de l'objet étudiant
    public static void SetName(Student student, string sName)
    {
      student.sName = sName;
    }
  }
}
```

Le programme crée un objet de la classe `Student`, qui ne comporte rien d'autre qu'un nom. Ici, nous aimons la simplicité chez les étudiants. Le programme commence par définir directement le nom de l'étudiant, et le

passe à la fonction d'affichage `OutputName()`, qui affiche alors le nom de tout objet de la classe `Student` qu'elle reçoit.

Le programme change alors le nom de l'étudiant en appelant la fonction `SetName()`. Comme en C#, tous les objets sont passés par référence, le changement fait à `student` est répercuté dans la fonction appelante. Lorsque `Main()` affiche à nouveau le nom de l'étudiant, celui-ci a changé :

```
La première fois :
Le nom de l'étudiant est Madeleine
Après avoir été modifié :
Le nom de l'étudiant est Willa
Appuyez sur Entrée pour terminer...
```

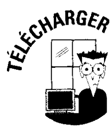
La fonction `SetName()` change le nom de l'objet `student` de la classe `Student`, et cette modification est reprise par le programme appelant.

Définir des fonctions et des méthodes d'objet

Une classe est faite pour rassembler des éléments qui représentent des objets ou des concepts du monde réel. Par exemple, une classe `Vehicle` peut contenir des éléments qui sont des données telles que la vitesse maximale, le poids, la capacité de charge, et ainsi de suite. Mais un objet de la classe `Vehicle` possède également des propriétés actives : la capacité de démarrer, de s'arrêter, et ainsi de suite. Celles-ci sont décrites par des fonctions qui utilisent les données des objets de cette classe. Ses fonctions font partie de la classe `Vehicle` tout autant que les propriétés de ses objets.

Définir une fonction membre statique d'une classe

Vous pourriez par exemple réécrire le programme de la section précédente en l'améliorant un peu :




```
// PassObjectToMemberFunction - utilise des fonctions membres statiques
//                              pour manipuler des champs
//                              dans l'objet
using System;
namespace PassObjectToMemberFunction
```

```
{
  public class Student
  {
    public string sName;
    // OutputName - affiche le nom de l'étudiant
    public static void OutputName(Student student)
    {
      //affiche le nom de l'étudiant courant
      Console.WriteLine("Le nom de l'étudiant est {0}", student.sName);
    }
    // SetName - modifie le nom de l'objet student
    public static void SetName(Student student, string sName)
    {
      student.sName = sName;
    }
  }
  public class Class1
  {
    public static void Main(string[] args)
    {
      Student student = new Student();
      //définit le nom en y accédant directement
      Console.WriteLine("La première fois :");
      student.sName = "Madeleine";
      Student.OutputName(student);
      //change le nom en utilisant une fonction
      Console.WriteLine("Après avoir été modifié :");
      Student.SetName(student, "Willa");
      Student.OutputName(student);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

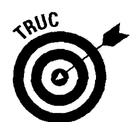
Ce programme ne présente qu'une différence significative avec le programme `PassObject` de la section précédente : j'ai mis les fonctions `OutputName()` et `SetName()` dans la classe `Student`.

Du fait de cette modification, `Main()` doit référencer la classe `Student` dans les appels à `SetName()` et à `OutputName()`. Ces deux fonctions sont maintenant des membres de la classe `Student`, et non de `Class1`, la fonction dans laquelle réside `Main()`.

C'est une étape modeste mais significative. Placer `OutputName()` dans la classe elle-même la rend plus réutilisable : une fonction extérieure qui aura

besoin d'afficher l'objet trouvera `OutputName()` avec d'autres fonctions d'affichage à cet endroit, car faisant partie de la classe.

C'est également une meilleure solution d'un point de vue philosophique. `Class1` ne doit pas avoir à se préoccuper de la manière d'initialiser le nom d'un objet `Student`, ni de la manière d'afficher des éléments importants. C'est la classe `Student` qui doit contenir ces informations.



En fait, ce n'est pas `Main()` qui devrait commencer par initialiser le nom à "Madeleine". Elle devrait plutôt appeler pour cela `SetName()`.

Depuis la classe `Student`, une fonction membre peut en invoquer une autre sans avoir à évoquer explicitement le nom de la classe. `SetName()` peut invoquer `OutputName()` sans avoir besoin pour cela de référencer le nom de la classe. Si vous omettez celui-ci, C# suppose que la fonction à laquelle vous voulez accéder appartient à la même classe.

Définir une méthode

C'est par l'objet, et non par la classe, que l'on accède à un membre donnée d'un objet. On peut donc écrire :

```
Student student = new Student();
student.sName = "Madeleine";
```

C# vous permet d'invoquer de la même manière une fonction membre non statique :

```
student.SetName("Madeleine");
```

C'est la technique que montre l'exemple suivant :




```
// InvokeMethod - invoque une fonction membre à partir de l'objet
using System;
namespace InvokeMethod
{
  class Student
  {
    // le nom de l'étudiant décrit l'objet student
    public string sFirstName;
    public string sLastName;
    // SetName - met de côté le nom de l'étudiant
```

```
    public void SetName(string sFName, string sLName)
    {
      sFirstName = sFName;
      sLastName  = sLName;
    }
    // ToNameString - converti en chaîne pour affichage
    //                l'objet student
    public string ToNameString()
    {
      string s = sFirstName + " " + sLastName;
      return s;
    }
  }
  public class Class1
  {
    public static void Main()
    {
      Student student = new Student();
      student.SetName("Stephen", "Davis");
      Console.WriteLine("Le nom de l'étudiant est "
                        + student.ToNameString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

La sortie de ce programme est cette simple ligne :

```
Le nom de l'étudiant est Stephen Davis
```

En dehors d'avoir un nom beaucoup plus court, ce programme est très semblable au programme `PassObjectToMemberFunction` que nous avons vu plus haut. Cette fonction utilise des fonctions non statiques pour manipuler un prénom et un nom.

Le programme commence par créer un nouvel objet, `student`, de la classe `Student`. Il invoque ensuite la fonction `SetName()`, qui stocke les deux chaînes "Stephen" et "Davis" dans les membres donnée `sFirstName` et `sLastName`. Enfin, le programme appelle la fonction membre `ToNameString()`, qui retourne le nom complet de `student` en concaténant les deux chaînes.

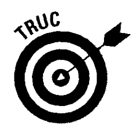


Pour des raisons historiques qui n'ont rien à voir avec C#, une fonction membre non statique est communément appelée une *méthode*. J'utilise le terme *méthode* pour une fonction membre non statique, et le terme *fonction* pour toutes les autres fonctions.

Regardez à nouveau la fonction `SetName()` qui met à jour le nom et le prénom dans un objet de la classe `Student`. Quel objet modifie `SetName()` ? Pour voir le problème, considérez l'exemple suivant :

```
Student christa = new Student();
Student sarah = new Student();
christa.SetName("Christa", "Smith");
sarah.SetName("Sarah", "Jones");
```

Le premier appel à `SetName()` met à jour le nom et le prénom de l'objet `christa`. Le deuxième appel met à jour l'objet `sarah`.



Voilà pourquoi un programmeur C# dit que cette méthode opère sur l'objet *courant*. Dans le premier appel, l'objet courant est `christa`, dans le deuxième, c'est `sarah`.

Pourquoi des méthodes ?

Pourquoi des méthodes ? Pourquoi de simples fonctions ne suffiraient-elles pas ? Les méthodes jouent deux rôles différents mais importants.

La méthode `SetName()` masque les détails de la manière dont les noms sont stockés dans la classe `Student`. Ce sont des informations dont des fonctions extérieures à `Student` ne sont pas censées avoir besoin. C'est un peu comme la manière dont nous utilisons les boutons d'un four à micro-ondes : ces boutons masquent le fonctionnement interne de l'appareil, que nous n'avons pas besoin de connaître.

Le second rôle d'une méthode est de représenter les propriétés véritables de la classe. Un avion peut accélérer, virer, décoller et atterrir (entre autres choses). Une classe `Airplane` complète devrait donc comporter les méthodes `Accelerate()`, `Bank()`, `TakeOff()`, et `Land()`, reproduisant fidèlement ces propriétés. Mettre la représentation d'une classe en accord avec son équivalent du monde réel permet de penser à un programme dans des termes qui sont ceux du véritable problème, plutôt qu'en un vocabulaire artificiel dicté par le langage de programmation utilisé.

Le nom complet d'une méthode

La description que j'ai faite du nom d'une méthode comporte un problème subtil mais important. Pour le voir, examinez l'exemple de code suivant :

```
public class Person
{
  public void Address()
  {
    Console.WriteLine("Hi");
  }
}
public class Letter
{
  string sAddress;
  //met de côté l'adresse
  public void Address(string sNewAddress)
  {
    sAddress = sNewAddress;
  }
}
```

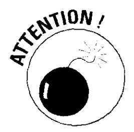
Toute considération ultérieure sur la méthode `Address()` est maintenant ambiguë. La méthode `Address()` de `Person` n'a rien à voir avec la méthode `Address()` de `Letter`. Si un ami programmeur me dit d'utiliser la méthode `Address()`, de quelle `Address()` parle-t-il ?

Le problème ne vient pas des méthodes elles-mêmes mais de ma description. En fait, il n'y a pas de méthode `Address()`, mais seulement une méthode `Person.Address()` et une méthode `Letter.Address()`. Ajouter le nom de la classe au début du nom de la méthode indique clairement de quelle méthode il s'agit.

Cette description est très semblable à la question des noms propres. Dans ma famille, on m'appelle Stephen. Il n'y a pas d'autre Stephen dans ma famille, mais il y en a deux autres là où je travaille.

Si je déjeune avec quelques collègues et que les deux autres Stephen ne sont pas là, il est évident que le nom *Stephen* se réfère à moi. Mais de retour dans les bureaux, si vous appelez le nom "Stephen", c'est ambigu car il peut se référer à n'importe lequel de nous trois. Il vous faudra donc appeler "Stephen Davis" pour éviter la confusion avec "Stephen Williams" ou "Stephen Leija".

Autrement dit, vous pouvez considérer `Address()` comme le prénom, ou le surnom, d'une méthode.



Le nom de la classe est un autre moyen de différencier des noms de méthode surchargés, les autres étant les noms et le nombre de ses arguments de fonction.

Accéder à l'objet courant

Examinez la méthode `Student.SetName()` suivante :

```
class Student
{
  //le nom de l'étudiant décrit l'objet student
  public string sFirstName;
  public string sLastName;
  // SetName - met de côté le nom de l'étudiant
  public void SetName(string sFName, string sLName)
  {
    sFirstName = sFName;
    sLastName  = sLName;
  }
}
public class Class1
{
  public static void Main()
  {
    Student student1 = new Student();
    student1.SetName("Joseph", "Smith");
    Student student2 = new Student();
    student2.SetName("John", "Davis");
  }
}
```

La fonction `Main()` utilise la méthode `SetName()` pour mettre à jour d'abord `student1`, puis `student2`. Mais vous ne voyez de référence à aucun objet de la classe `Student` dans la méthode `SetName()` elle-même. En fait, elle ne contient aucune référence à un objet de la classe `Student`. Une méthode opère sur "l'objet courant". Comment fait-elle pour savoir quel est l'objet courant ? L'objet courant est prié de se lever.

La réponse est simple. L'objet courant est passé comme argument implicite dans l'appel à la méthode. Par exemple :

```
student1.SetName("Joseph", "Smith");
```

Cet appel est équivalent à :

```
Student.SetName(student1, "Joseph", "Smith"); //appel équivalent
                                    // (mais ceci ne sera pas
                                    // généré correctement)
```

Je ne suis pas en train de dire que vous pouvez invoquer `SetName()` de deux manières différentes, mais simplement que les deux appels sont équivalents d'un point de vue sémantique. L'objet qui se trouve juste à gauche de "." (le premier argument caché) est passé à la fonction tout comme les autres arguments.

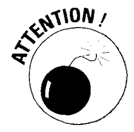
Passer un objet implicitement est facile à avaler, mais que diriez-vous d'une référence d'une méthode à une autre ?

```
public class Student
{
  public string sFirstName;
  public string sLastName;
  public void SetName(string sFirstName, string sLastName)
  {
    SetFirstName(sFirstName);
    SetLastName(sLastName);
  }
  public void SetFirstName(string sName)
  {
    sFirstName = sName;
  }
  public void SetLastName(string sName)
  {
    sLastName = sName;
  }
}
```

Aucun objet n'apparaît dans l'appel à `SetFirstName()`. L'objet courant continue à être passé en silence d'un appel de méthode au suivant. Un accès à n'importe quel membre depuis une méthode d'objet est censé concerner l'objet courant.

Qu'est-ce que `this` ?

Contrairement à la plupart des arguments, toutefois, l'objet courant n'apparaît pas dans la liste des arguments de la fonction et ne se voit donc pas assigner un nom par le programmeur. Au lieu de cela, C# assigne à cet objet le nom `this`.



`this` est un mot-clé, et ne doit donc être utilisé pour rien d'autre.

On pourrait donc écrire l'exemple précédent de la façon suivante :

```
public class Student
{
  public string sFirstName;
  public string sLastName;
  public void SetName(string sFirstName, string sLastName)
  {
    // référence explicitement "l'objet courant" référencés par this
    this.SetFirstName(sFirstName);
    this.SetLastName(sLastName);
  }
  public void SetFirstName(string sName)
  {
    this.sFirstName = sName;
  }
  public void SetLastName(string sName)
  {
    this.sLastName = sName;
  }
}
```

Remarquez l'introduction explicite du mot-clé `this`. Ajouter `this` aux références au membre n'ajoute en fait rien, car `this` est implicitement supposé. Toutefois, quand `Main()` effectue l'appel suivant, `this` référence `student1` partout dans `SetName()` et dans toute autre méthode que pourrait appeler `Main()`.

```
student1.SetName("John", "Smith");
```

Quand `this` est-il explicite ?

Normalement, il n'est pas nécessaire de se référer explicitement à `this`, car il est implicitement compris par le compilateur là où il est nécessaire. Toutefois, il y a deux cas assez courants dans lesquels `this` est nécessaire. Tout d'abord, pour initialiser un membre donnée :

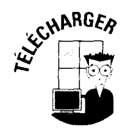
```
// Address - définit un "cadre de base" pour une adresse aux USA
class Person
{
  public string sName;
```

```
    public int nID;
    public void Init(string sName, int nID)
    {
      this.sName = sName;
      this.nID = nID;
    }
  }
```

Les arguments de la méthode `Init()` sont nommés `sName` et `nID`, ce qui est identique aux noms des membres donnée correspondants. Cette disposition rend la fonction facile à lire, car elle permet de savoir exactement où est stocké quel argument. Le seul problème est que le nom `sName` dans la liste des arguments rend obscur le nom du membre donnée.

L'introduction de `this` permet de savoir de quel `sName` il s'agit : dans `Init()`, le nom `sName` se réfère à l'argument de la fonction, alors que `this.sName` se réfère au membre donnée.

Vous aurez également besoin de `this` pour mettre de côté l'objet courant afin de l'utiliser ultérieurement ou de le faire utiliser par une autre fonction. Regardez l'exemple suivant, `ReferencingThisExplicitly`:




```
// ReferencingThisExplicitly - ce programme montre
//            comment utiliser explicitement la référence à this
using System;
namespace ReferencingThisExplicitly
{
  public class Class1
  {
    public static int Main(string[] strings)
    {
      //crée un objet student
      Student student = new Student();
      student.Init("Stephen Davis", 1234);
      //inscrit l'étudiant à un cours
      Console.WriteLine
            ("Inscription de Stephen Davis à Biologie 101");
      student.Enroll("Biologie 101");
      //affichage des cours auxquels est inscrit l'étudiant
      Console.WriteLine("Nouvelles caractéristiques de l'étudiant :");
      student.DisplayCourse();
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
```

```
  // Student - notre étudiant d'université
  public class Student
  {
    //tout étudiant a un nom et un numéro d'identification (id)
    public string sName;
    public int    nID;
    //le cours auquel est inscrit l'étudiant
    CourseInstance courseInstance;
    // Init - initialise l'objet student
    public void Init(string sName, int nID)
    {
      this.sName = sName;
      this.nID = nID;
      courseInstance = null;
    }
    // Enroll - inscrit l'étudiant courant à un cours
    public void Enroll(string sCourseID)
    {
      courseInstance = new CourseInstance();
      courseInstance.Init(this, sCourseID);
    }
    //affiche le nom de l'étudiant
    //et le cours
    public void DisplayCourse()
    {
      Console.WriteLine(sName);
      courseInstance.Display();
    }
  }
  // CourseInstance - associe l'étudiant au cours
  //                  auquel il est inscrit
  public class CourseInstance
  {
    public Student student;
    public string sCourseID;
    // Init - établit le lien entre l'étudiant et le cours
    public void Init(Student student, string sCourseID)
    {
      this.student = student;
      this.sCourseID = sCourseID;
    }
    // Display - affiche l'intitulé du cours
    public void Display()
    {
      Console.WriteLine(sCourseID);
    }
  }
} }
```

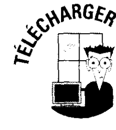
Ce programme est très quelconque. L'objet de la classe `Student` peut contenir un nom, un identificateur, et un seul type de cours universitaire (ce n'est pas un étudiant très occupé). `Main()` crée l'étudiant, puis invoque `Init()` pour initialiser l'objet de la classe `Student`. À ce point, la référence `courseInstance` reçoit la valeur `null`, car l'étudiant ne s'est pas encore inscrit au cours.

La méthode `Enroll()` inscrit l'étudiant en initialisant `courseInstance` avec un nouvel objet. Toutefois, la méthode `CourseInstance.Init()` prend un étudiant comme premier argument avec l'identificateur du cours comme deuxième argument. Quel étudiant faut-il passer ? Il est évident qu'il faut passer l'étudiant courant, celui auquel se réfère `this` (on peut donc dire que `Enroll()` inscrit cet (`this`) étudiant au cours `CourseInstance`). Les méthodes `Display()` affichent l'étudiant et les noms des cours.

Et quand je n'ai pas `this` *?*

Mélanger des fonctions de classe et des méthodes d'objet, c'est un peu comme de mélanger des cow-boys et les propriétaires de ranch. Heureusement, C# vous donne quelque moyen de contourner les problèmes relationnels de ces créatures. Ça me rappelle un peu la chanson d'*Oklahoma!* : "Oh, la fonction et la méthode peuvent être amies..."

Pour voir le problème, regardez l'exemple de programme `MixingFunctionsAndMethods` :




```
// MixingFunctionsAndMethods - mélanger des fonctions de classe et
//                    des méthodes d'objet peut causer des problèmes
using System;
namespace MixingFunctionsAndMethods
{
  public class Student
  {
    public string sFirstName;
    public string sLastName;
    // InitStudent - initialise l'objet student
    public void InitStudent(string sFirstName, string sLastName)
    {
      this.sFirstName = sFirstName;
      this.sLastName = sLastName;
    }
    // OutputBanner - affiche l'introduction
    public static void OutputBanner()
    {
      Console.WriteLine("Regardez comme je suis malin :");
```

```
      // Console.WriteLine(? quel objet student utilisons-nous ?);
    }
    public void OutputBannerAndName()
    {
      // c'est la classe Student qui est supposée mais pas ça
      // l'objet est passé à la méthode statique
      OutputBanner();
      // ce n'est pas l'objet this qui est passé mais l'objet
      // student courant qui est passé explicitement
      OutputName(this);
    }
    // OutputName - affiche le nom de l'étudiant
    public static void OutputName(Student student)
    {
      // ici, l'objet student est référencé explicitement
      Console.WriteLine("Le nom de l'étudiant est {0}",
                        student.ToNameString());
    }
    // ToNameString - va chercher le nom de l'étudiant
    public string ToNameString()
    {
      // ici, le nom de l'objet courant est implicite -
      // ce qui aurait pu être écrit :
      // return this.sFirstName + " " + this.sLastName;
      return sFirstName + " " + sLastName;
    }
  }
  public class Class1
  {
    public static void Main(string[] args)
    {
      Student student = new Student();
      student.InitStudent("Madeleine", "Cather");
      // affiche la bannière et le nom
      Student.OutputBanner();
      Student.OutputName(student);
      Console.WriteLine();
      // affiche à nouveau la bannière et le nom
      student.OutputBannerAndName();
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

Commencez par le bas, avec `Main()`, pour mieux voir le problème. Ce programme commence par créer un objet `Student` et initialiser son nom.

Maintenant, ce nigaud (le programme, pas l'étudiant) veut simplement afficher le nom, précédé par un bref message et une bannière.

`Main()` commence par afficher la bannière et le message en utilisant des fonctions de classe. Le programme invoque la fonction `OutputBanner()` pour la bannière, et la fonction `OutputName()` pour afficher le message et le nom de l'étudiant. La fonction `OutputBanner()` affiche simplement un message sur la console. `Main()` passe l'objet `student` comme argument à `OutputName()` afin que celle-ci puisse afficher le nom de l'étudiant.

Ensuite, `Main()` utilise l'approche de la fonction ou de la méthode d'objet pour afficher la bannière et le message en appelant `student.OutputBannerAndName()`.

`OutputBannerAndName()` commence par invoquer la fonction statique `OutputBanner()`. La classe `Student` est supposée. Aucun objet n'est passé, car la fonction statique n'en a pas besoin. Ensuite, `OutputBannerAndName()` appelle la fonction `OutputName()`. Celle-ci est également une fonction statique mais un objet de la classe `Student` lui est passé comme argument par `OutputBannerAndName()`.

Un cas plus intéressant est l'appel de `ToNameString()` depuis `OutputName()`. Cette dernière fonction est déclarée `static`, et par conséquent n'a pas de `this`. Elle a un objet explicite de la classe `Student` qu'elle utilise pour réaliser cet appel.

La fonction `OutputBanner()` voudrait peut-être pouvoir appeler aussi `ToNameString()`, mais elle n'a pas d'objet de la classe `Student` à utiliser. Elle n'a pas de pointeur `this` parce que c'est une fonction statique et qu'aucun objet ne lui a été passé explicitement.



Une fonction statique ne peut pas appeler une méthode non statique sans lui passer explicitement un objet. Pas d'objet, pas d'appel.

Obtenir de l'aide de Visual Studio – la saisie automatique

Visual Studio .NET comporte une fonction de saisie automatique extrêmement utile au programmeur. Lorsque vous tapez le nom d'une classe ou d'un objet dans votre code source, Visual Studio utilise les premiers caractères que vous tapez pour anticiper la suite et vous proposer un choix de noms parmi lesquels se trouve celui que vous voulez saisir.

Cette fonction de saisie automatique est plus facile à décrire par un exemple. J'utiliserai pour cela le fragment suivant du code source du programme `MixingFunctionsAndMethods` :

```
// affiche la bannière et le nom
Student.OutputBanner();
Student.OutputName(student);
Console.WriteLine();
// affiche à nouveau la bannière et le nom
student.OutputBannerAndName();
```

Obtenir de l'aide sur les fonctions intégrées de la bibliothèque standard C#

Dans le fragment de code ci-dessus, lorsque je tape **Console.**, Visual Studio affiche la liste des méthodes de `Console`. Lorsque que je tape le `W`, Visual Studio sélectionne dans cette liste la première méthode dont le nom commence par *W*, qui est `Write()`. Le déplacement de la sélection d'un cran vers le bas, en utilisant la touche de curseur correspondante, sélectionne `WriteLine()`. À droite de la liste, en regard de `WriteLine()`, apparaît une info-bulle qui en contient la description, comme le montre la Figure 8.1. Cette info-bulle indique également qu'il existe dix-huit autres versions surchargées de la fonction `WriteLine()` (chacune avec un ensemble d'arguments différent, bien sûr).

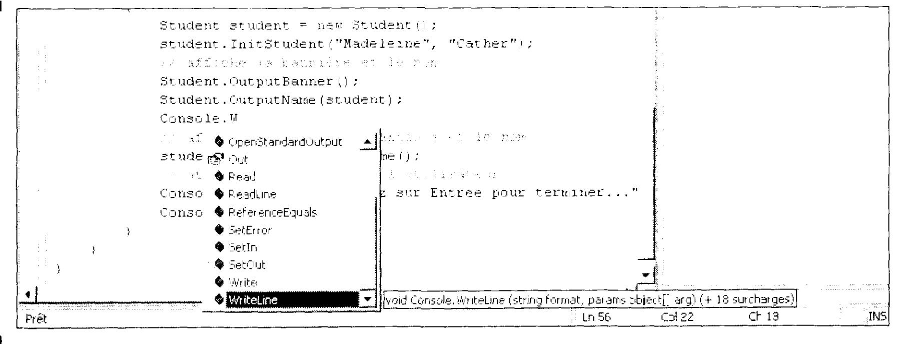


Figure 8.1 : La fonction de saisie automatique de Visual Studio est une aide précieuse pour choisir la bonne méthode.

Il me reste à compléter le nom de la fonction, `WriteLine`. Dès que je tape la parenthèse ouvrante, Visual Studio affiche une info-bulle indiquant les arguments que comporte la fonction, comme le montre la Figure 8.2. Remarquez

que cette info-bulle commence par indiquer le numéro de la version de la fonction parmi toutes celles qui existent, avec deux flèches qui permettent de faire défiler ces versions pour identifier celle que vous voulez.

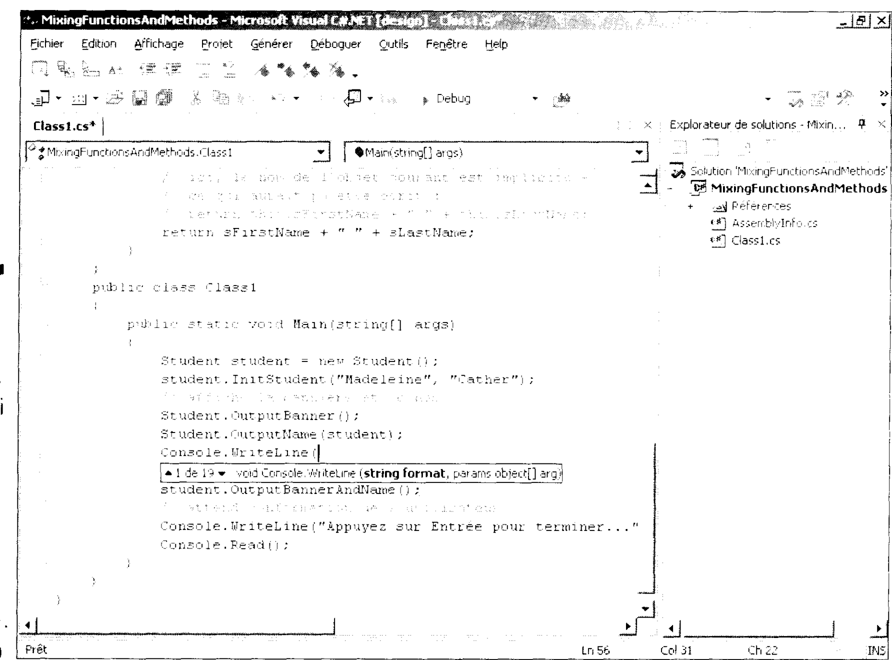
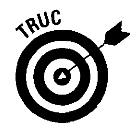
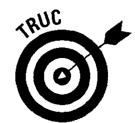


Figure 8.2 : La fonction de saisie automatique affiche aussi la liste des arguments pour la version de votre choix de la fonction `WriteLine()`.



Vous n'avez donc pas besoin de taper le nom de la fonction. Imaginez que vous ayez tapé **WriteL** pour identifier exactement la méthode voulue. En voyant le nom `WriteLine` sélectionné dans la liste, il vous suffit de taper une parenthèse ouvrante pour que Visual Studio complète automatiquement ce nom pour vous, après quoi, il vous restera à taper les paramètres que vous voulez passer, et la parenthèse fermante.



Pour faire apparaître la description des arguments de la version de `WriteLine()` que vous cherchez, cliquez sur l'une des flèches dans l'info-bulle qui apparaît lorsque vous tapez la parenthèse ouvrante. Dans cette info-bulle, la description du premier argument que vous avez à saisir apparaît en gras, comme le montre la Figure 8.2.

Aussitôt que j'ai entré la chaîne "chaîne", et une virgule, Visual Studio met en gras la description du prochain argument à saisir, comme le montre la Figure 8.3.

Figure 8.3 : À chaque étape, Visual Studio affiche en gras la description du prochain argument à saisir.

Dès que vous tapez la virgule qui suit un argument après l'avoir saisi, Visual Studio affiche en gras dans l'info-bulle la description du prochain argument à saisir. Bien sûr, cette aide est disponible pour toutes les méthodes intégrées de la bibliothèque standard C# utilisées par votre programme.

Obtenir de l'aide sur vos propres fonctions et méthodes

Vous pouvez aussi obtenir de l'aide sur vos propres fonctions.

En continuant avec l'exemple de la section précédente, j'efface la chaîne "chaîne" pour la remplacer intentionnellement par une chaîne vide : `Console.WriteLine()`. Sur la ligne suivante, je tape "**student.**". Dès que j'ai tapé le point, Visual Studio affiche la liste des membres de l'objet `student`, comme le montre la Figure 8.4.

Remarquez les icônes qui précèdent les noms des méthodes dans la fenêtre d'aide : un petit rectangle qui penche vers la droite indique un membre donnée ; un petit rectangle qui penche vers la gauche indique une méthode.

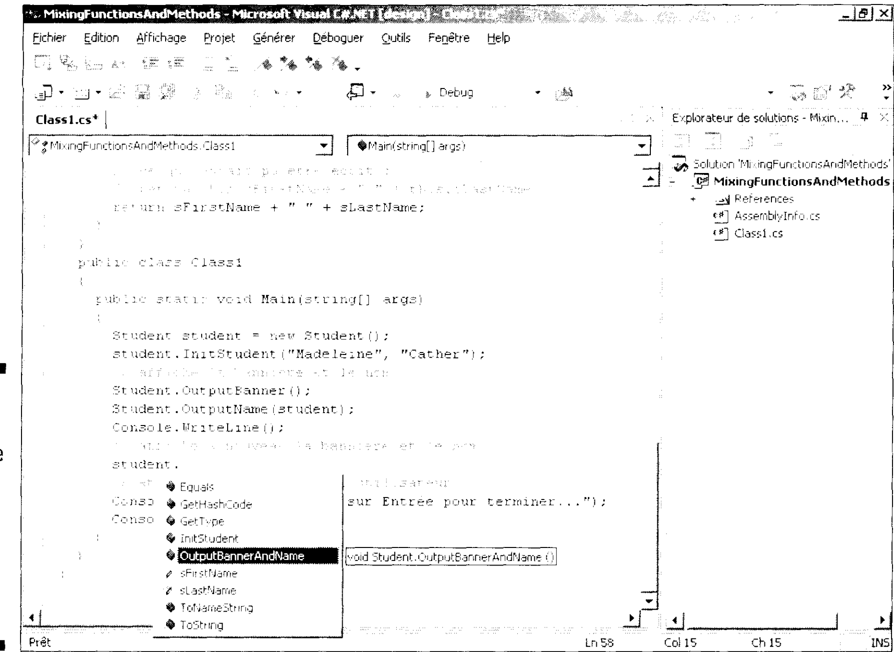
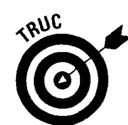
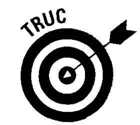


Figure 8.4 : La saisie automatique est également disponible pour vos propres méthodes.



Ces icônes sont faciles à reconnaître. Celle d'un membre donnée est en bleu clair, celle d'une méthode est en violet et précédée de trois traits horizontaux.

Dans la fenêtre, il y a des méthodes que je ne reconnais pas. Ce sont des méthodes de base que reçoivent d'office tous les objets. Dans ce groupe de méthodes standard, vous voyez notre propre `OutputBannerAndName()`. Dès que je tape le O, elle est mise en surbrillance, et l'info-bulle apparaît pour en décrire les arguments, afin que je sache comment l'utiliser.



Encore une fois, il vous suffit alors de taper une parenthèse ouvrante pour que le nom de la méthode, préalablement mis en surbrillance dans la liste, soit automatiquement complété.

Cette aide marche aussi pour les fonctions. Lorsque j'entre le nom de classe `Student` suivi par un point, Visual Studio affiche la liste des membres de `Student`. Si je tape ensuite **OutputN**, Visual Studio affiche l'info-bulle contenant la liste des arguments de `OutputName()`, comme le montre la Figure 8.5.

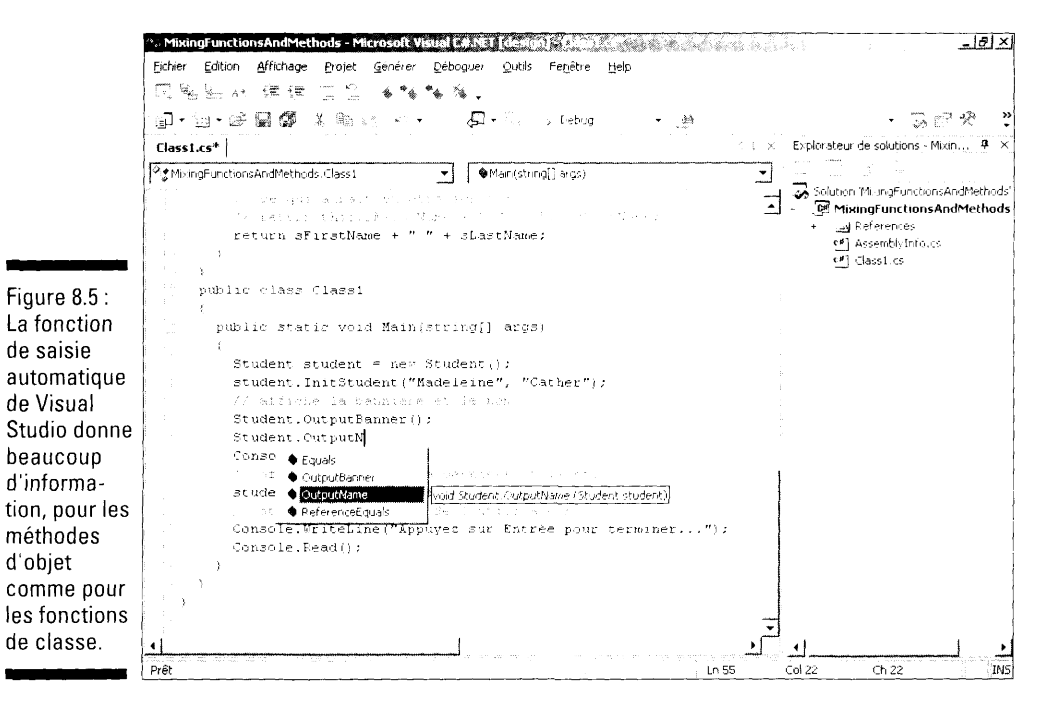


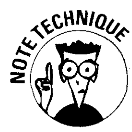
Figure 8.5 : La fonction de saisie automatique de Visual Studio donne beaucoup d'information, pour les méthodes d'objet comme pour les fonctions de classe.

Encore plus d'aide

La fonction de saisie automatique de Visual Studio apporte une aide importante en anticipant sur les membres que vous voulez saisir dès que vous entrez le nom de la classe ou de l'objet.

Visual Studio ne peut fournir qu'une aide limitée pour les fonctions et les classes créées par l'utilisateur. Par exemple, il ne sait pas ce que fait la méthode `OutputName()`. Heureusement, Visual Studio vous offre un moyen détourné de dire à la fonction de saisie automatique ce que fait la fonction, et même un peu plus.

Pour indiquer une ligne de commentaire normal, vous utilisez deux barres obliques : //. Mais Visual Studio comprend aussi comme un commentaire spécial ce qui est indiqué par trois barres obliques : ///. Un tel *commentaire de documentation* permet de donner à Visual Studio des informations supplémentaires, utilisables par la fonction de saisie automatique.

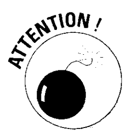


Pour être honnête, c'est le langage Java qui a introduit cette idée. Java dispose d'un programme supplémentaire capable d'extraire les commentaires marqués par ces trois barres obliques pour les rassembler dans un fichier de documentation séparé. C# a apporté une amélioration à cette innovation : l'aide en cours d'édition.

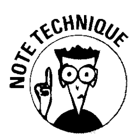
Un commentaire de documentation peut contenir n'importe quelle combinaison des commandes montrées par le Tableau 8.1.

Tableau 8.1 : Balises communes des commentaires de documentation.

Balise	Signification
<param></param>	Description d'un argument de la fonction, affichée par l'aide après la saisie du nom de la fonction et de la parenthèse ouvrante, expliquant ce que vous avez à saisir.
<summary></summary>	Description de la fonction elle-même, affichée en cours d'édition lors de la saisie du nom de la fonction.
<returns></returns>	Description de la valeur retournée par la fonction.



Un commentaire de documentation doit se conformer à la règle XML/HTML : une commande commence par `<command>` et se termine par `</command>`. En fait, on les appelle ordinairement *balises XML*, du fait de leur relation avec XML.



Vous disposez de bien d'autres balises XML. Pour en savoir plus à leur sujet, consultez l'aide en ligne de Visual Studio (plus officiellement connue sous le nom de MSDN pour Visual Studio) en sélectionnant ?/Index, et tapez "XML" dans le champ Rechercher.

L'exemple suivant est une version commentée du programme `MixingFunctionsAndMethods` :




```
// MixingFunctionsAndMethods - mélanger des fonctions de classe et des
//                          méthodes d'objet peut causer des problèmes
using System;
namespace MixingFunctionsAndMethods
{
  /// <summary>
  /// simple description d'un étudiant
```

```
        thisStudent.dGPA  = dGPA;
        // ajoute l'objet Student au tableau
        students[i] = thisStudent;
      }
      // calcule la moyenne des étudiants du tableau
      double dSum = 0.0;
      for (int i = 0; i < students.Length; i++)
      {
        dSum += students[i].dGPA;
      }
      double dAvg = dSum/students.Length;
      // output the average
      Console.WriteLine();
      Console.WriteLine("La moyenne générale des "
                       + students.Length
                       + " étudiants est " + dAvg);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

Le programme demande à l'utilisateur le nombre d'étudiants à prendre en compte. Il crée ensuite le tableau de références à des objets `Student`, correctement dimensionné.

Le programme entre maintenant dans une boucle `for` initiale qui va lui permettre de remplir le tableau. L'utilisateur se voit demander le nombre et la moyenne des UV de chaque étudiant, l'un après l'autre. Ces données sont utilisées pour créer un objet de `Student`, qui devient aussitôt le nouvel élément du tableau.

Une fois que toutes les références à des objets de `Student` sont à leur place, le programme entre dans une deuxième boucle. Dans celle-ci, la moyenne des UV de chaque étudiant est lue au moyen de l'instruction `students[i].GPA`. Toutes ces moyennes sont arrondies et additionnées, la moyenne générale en est calculée, puis finalement affichée pour l'utilisateur.

Voici un exemple de résultats affichés par ce programme :

```
Entrez le nombre d'étudiants
3
Entrez le nom de l'étudiant 1: Randy
Entrez sa moyenne de points d'UV : 3.0
Entrez le nom de l'étudiant 2: Jeff
```

Des tableaux d'objets

Les programmeurs ont souvent besoin de travailler avec des ensembles d'objets définis par l'utilisateur. Par exemple, une université aura besoin de définir une structure pour décrire la population des étudiants qui suivent ses cours.

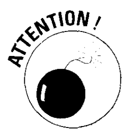
Une classe `Student` simplifiée peut se définir ainsi :

```
public class Student
{
  public string sName;
  public double dGPA;          // moyenne des points d'UV
}
```

Cette classe ne contient rien d'autre que le nom de l'étudiant et la moyenne des points de ses "unités de valeur" (ou UV). Je mets "unités de valeur" entre guillemets parce que cet exemple (et tous ceux qui suivent jusqu'à la fin du chapitre) repose sur le système universitaire américain, dans lequel la notion de "grade" correspond très approximativement à nos UV (GPA signifie Grade Point Average, autrement dit, dans notre exemple, *moyenne des points d'UV*).

La ligne suivante déclare un tableau de `num` références à des objets de la classe `Student` :

```
Student[] students = new Student[num];
```



`new Student[num]` ne déclare *pas* un tableau d'objets de la classe `Student`. Cette ligne déclare un tableau de références à des objets de la classe `Student`.

Jusqu'ici, chaque élément `students[i]` référence l'objet `null`. On pourrait aussi dire qu'aucun des éléments du tableau ne pointe vers un objet de `Student`. Il faut commencer par remplir le tableau, comme ceci :

```
for (int i = 0; i < students.Length; i++)
{
    students[i] = new Students();
}
```

```
Entrez la valeur nº5: 5

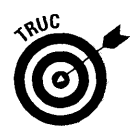
3 est la moyenne de (1 + 2 + 3 + 4 + 5) / 5
Appuyez sur Entrée pour terminer...
```

Le programme `VariableArrayAverage` commence par demander à l'utilisateur le nombre de valeurs dont il veut calculer la moyenne. Le résultat est stocké dans la variable `int numElements`. Dans l'exemple ci-dessus, j'ai entré la valeur 5.

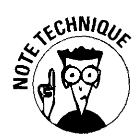
Le programme continue en définissant le tableau `dArray` avec le nombre d'éléments spécifié. Dans ce cas, il définit un tableau à cinq éléments. Puis le programme effectue une boucle avec le nombre d'itérations spécifié par `numElements`, lisant chaque fois une nouvelle valeur entrée par l'utilisateur.

Une fois que l'utilisateur a entré les valeurs, le programme applique le même algorithme utilisé par le programme `FixedArrayAverage` pour calculer la moyenne des valeurs.

Enfin, la section finale affiche le résultat du calcul, avec les valeurs qui ont été entrées, dans une présentation agréable à lire.



Il n'est pas toujours facile d'obtenir un affichage satisfaisant sur la console. Examinez soigneusement chaque instruction du programme `FixedArrayAverage`, les accolades ouvrantes, les signes égale, les signes plus, et toutes les valeurs de la séquence, et comparez le tout avec l'affichage.



Le programme `VariableArrayAverage` ne satisfait pas entièrement ma soif de souplesse. Je ne veux pas avoir besoin de lui dire de combien de valeurs je veux faire la moyenne. Je préfère entrer autant de nombres que je veux, et demander au programme au moment que je choisis de calculer la moyenne de ce que j'ai entré. C# offre d'autres types de conteneurs, dont certains que je peux agrandir ou réduire à volonté. Ils sont décrits au Chapitre 16.

La propriété `Length`

La boucle `for` que nous avons utilisée pour remplir le tableau dans le programme `VariableArrayAverage` commence de la façon suivante :

```
// déclare un tableau de la taille correspondante
double[] dArray = new double[numElements];
// remplit le tableau avec les valeurs
for (int i = 0; i < numElements; i++)
```

Le tableau à longueur variable

Le tableau utilisé dans l'exemple de programme `FixedArrayAverage` souffre de deux problèmes sérieux. Tout d'abord, la taille du tableau est fixée à dix éléments. Pire encore, la valeur de ces dix éléments est directement spécifiée dans le programme.

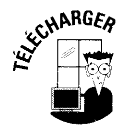
Un programme qui pourrait lire un nombre variable de valeurs, éventuellement déterminées par l'utilisateur au cours de l'exécution, serait beaucoup plus souple. Il fonctionnerait non seulement pour les dix valeurs spécifiées dans `FixedArrayAverage`, mais aussi pour n'importe quel autre ensemble de valeurs.

La déclaration d'un tableau de longueur variable diffère légèrement de celle d'un tableau de longueur fixe et à valeurs fixes :

```
double[] dArray = new double[N];
```

`N` représente le nombre d'éléments à allouer.

La nouvelle version de ce programme, `VariableArrayAverage`, permet à l'utilisateur de spécifier le nombre de valeurs à entrer. Comme ce programme conserve les valeurs entrées, non seulement il calcule la moyenne, mais il affiche aussi les résultats dans un format agréable :




```
// VariableArrayAverage - fait la moyenne des valeurs
//                        d'un tableau dont la taille est déterminée
//                        par l'utilisateur lors de l'exécution.
//                        Remplir un tableau avec des valeurs
//                        permet de les référencer aussi souvent
//                        que l'on veut. Dans ce cas, le tableau
//                        produit un affichage agréable.
namespace VariableArrayAverage
{
  using System;
  public class Class1
  {
    public static int Main(string[] args)
    {
      // commence par lire le nombre de types double
      // que l'utilisateur a l'intention d'entrer
      Console.Write("Nombre de valeurs pour la moyenne à calculer : ");
      string sNumElements = Console.ReadLine();
      int numElements = Convert.ToInt32(sNumElements);
      Console.WriteLine();
```

```
/// </summary>
public class Student
{
  /// <summary>
  /// l'étudiant reçoit un nom
  /// </summary>
  public string sFirstName;
  /// <summary>
  /// nom de famille de l'étudiant
  /// </summary>
  public string sLastName;

  // InitStudent - initialise l'objet student
  /// <summary>
  /// initialise l'objet student avant qu'il puisse être utilisé
  /// </summary>
  /// <param name="sFirstName">l'étudiant reçoit un nom</param>
  /// <param name="sLastName">nom de famille de l'étudiant</param>
  public void InitStudent(string sFirstName, string sLastName)
  {
    this.sFirstName = sFirstName;
    this.sLastName = sLastName;
  }
  // OutputBanner - affiche l'introduction
  /// <summary>
  /// affiche une bannière avant d'afficher les noms des étudiants
  /// </summary>
  public static void OutputBanner()
  {
    Console.WriteLine("Regardez comme je suis malin :");
    // Console.WriteLine(? quel objet student utilisons-nous ?);
  }

  // OutputBannerAndName
  /// <summary>
  /// affiche une bannière suivie par le nom de l'objet student courant
  /// </summary>
  public void OutputBannerAndName()
  {
    // c'est la classe Student qui est supposée mais pas ça
    // l'objet est passé à la méthode statique
    OutputBanner();

    // ce n'est pas l'objet this qui est passé mais l'objet
    // student courant qui est passé explicitement
    OutputName(this, 5);
  }
```

```
// OutputName - affiche le nom de l'étudiant
/// <summary>
/// affiche sur la console le nom de l'étudiant
/// </summary>
/// <param name="student">Le nom de l'étudiant que
///                       vous voulez afficher</param>
/// <param name="nIndent">Nombre d'espaces pour le retrait</param>
/// <returns>La chaîne qui a été affichée</returns>
public static string OutputName(Student student,
                                int nIndent)
{
  // ici, l'objet student est référencé explicitement
  string s = new String(' ', nIndent);
  s += String.Format("Le nom de l'étudiant est {0}",
                     student.ToNameString());
  Console.WriteLine(s);
  return s;
}

// ToNameString - va chercher le nom de l'étudiant
/// <summary>
/// convertit en chaîne le nom de l'étudiant pour l'afficher
/// </summary>
/// <returns>Le nom de l'étudiant sous forme de chaîne</returns>
public string ToNameString()
{
  // ici, le nom de l'objet courant est implicite -
  // ce qui aurait pu être écrit :
  // return this.sFirstName + " " + this.sLastName;
  return sFirstName + " " + sLastName;
}
}

/// <summary>
/// Exercise class
/// </summary>
public class Class1
{
  /// <summary>
  /// Le programme commence ici
  /// </summary>
  /// <param name="args"></param>
  public static void Main(string[] args)
  {
    Student student = new Student();
    student.InitStudent("Madeleine", "Cather");
```

```
            // affiche la bannière et le nom
            Student.OutputBanner();
            string s = Student.OutputName(student, 5);
            Console.WriteLine();

            // affiche à nouveau la bannière et le nom
            student.OutputBannerAndName();

            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
        }
    }
}
```

Les commentaires expliquent la finalité de la fonction, à quoi sert chaque argument, le type de donnée retournée, et la relation avec une autre fonction.

En pratique, les étapes suivantes décrivent ce qui est affiché lorsque je saisis dans `Main()` la fonction `Student.OutputName()` :

1. Visual Studio me propose une liste de fonctions. Une fois que j'ai mis en surbrillance celle que je veux, `OutputName()`, Visual Studio en affiche une courte description, extraite de `<summary></summary>`, comme le montre la Figure 8.6.

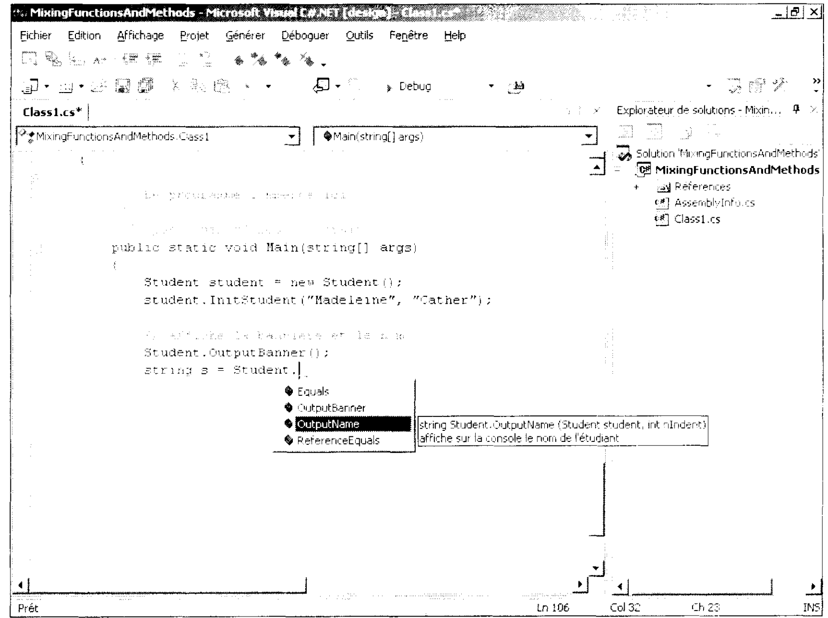


Figure 8.6 : Avec un programme documenté en XML, Visual Studio est capable de bien mieux décrire la fonction et ses arguments.

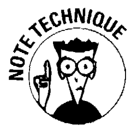
2. Une fois que j'ai saisi ou sélectionné le nom de la fonction, Visual Studio affiche une description du premier paramètre, extraite du champ `<param></param>`, ainsi que son type.

3. Visual Studio répète ce processus pour le deuxième argument, `nIndent`.

Bien qu'ils soient un peu fastidieux à saisir, les commentaires de documentation rendent les méthodes beaucoup plus faciles à utiliser.

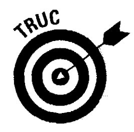
Générer une documentation XML

Vous pouvez facilement demander à Visual Studio d'extraire sous forme de fichier XML tous les commentaires de documentation que vous avez entrés.



Cette section est très technique. Si vous ne savez pas ce qu'est un fichier XML, tout cela ne vous dira pas grand-chose. Si vous savez comment est fait un fichier XML, vous allez trouver cette fonction très utile.

Sélectionnez Affichage/Explorateur de solutions pour afficher l'Explorateur de solutions. Dans l'Explorateur de solutions, cliquez du bouton droit sur le nom du programme, et sélectionnez Propriétés. Dans le volet de gauche de la fenêtre Pages de propriétés, cliquez sur le dossier Propriétés de configuration pour l'ouvrir, et sélectionnez Générer dans les pages qui apparaissent au-dessous de ce dossier. Dans la section Sortie du volet de droite de la fenêtre Pages de propriétés, sélectionnez la propriété nommée `Fichier de documentation XML`. Dans la cellule qui se trouve à droite de ce nom, entrez un nom de fichier. Comme je n'avais pas de meilleure idée, j'ai mis `xmloutput.xml`. Cliquez sur OK pour appliquer cette modification et fermer la fenêtre Pages de propriétés.



Vous pouvez aussi accéder aux propriétés du projet en sélectionnant Projet/Propriétés.

Sélectionnez maintenant Générer/Régénérer tout pour être sûr que tout a bien été généré correctement.

Regardez dans le même dossier que le fichier source `Class1.cs` (le fichier du projet est dans le même dossier). Le nouveau fichier `xmloutput.xml` décrit toutes les fonctions documentées par les balises XML.

Chapitre 9

Jouer avec des chaînes en C#

Dans ce chapitre :

Tordre une chaîne et tirer dessus – mais vous ne pouvez pas la pousser.

Analyser une chaîne lue par le programme.

Mettre en forme manuellement une chaîne de sortie.

Mettre en forme une chaîne de sortie en utilisant la méthode `String.Format()`.

Pour de nombreuses applications, vous pouvez traiter un élément de type `string` comme n'importe quel type de variable intégré, tel que `int` ou `char`. Certaines des opérations ordinairement réservées pour ces types intrinsèques sont utilisables pour les chaînes :

```
int i = 1;          // déclare et initialise un int
string s = "abc";   // déclare et initialise un string
```

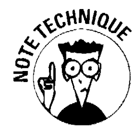
Pour d'autres aspects, un élément `string` est traité comme une classe définie par l'utilisateur :

```
string s1 = new String();
string s2 = "abcd";
int nLengthOfString = s2.Length;
```

Alors, qu'est-ce que c'est : un type de variable ou une classe ? En fait, `String` est une classe pour laquelle C# offre un traitement spécial. Par exemple, le mot-clé `string` est synonyme du nom de classe `String` :

```
String s1 = "abcd"; // assigne une chaîne littérale à un objet String
string s2 = s1;     // assigne un objet String à une variable string
```

Dans cet exemple, s1 est déclaré en tant qu'objet de la classe String (avec un *S* majuscule), alors que s2 est déclaré en tant que variable de type string (avec un *s* minuscule). Mais ces deux assignations montrent que string et String sont de même type (autrement dit, compatibles).



En fait, cette propriété est également vraie pour les autres types de variable, mais dans une mesure plus limitée. Même le type int possède sa classe correspondante, Int32, double correspond à la classe Double, et ainsi de suite. La différence est que string et String sont réellement une seule et même chose.

Effectuer des opérations courantes sur une chaîne

Les programmeurs C# effectuent plus d'opérations sur les chaînes que la chirurgie esthétique sur les hollywoodiens qui ne demandent que ça. Il n'y a guère de programmes qui n'utilisent pas l'opérateur d'addition sur des chaînes :

```
string sName = "Randy";
Console.WriteLine("Son nom est " + sName);
```

C'est la classe String qui fournit cet opérateur spécial, mais elle offre également d'autres méthodes, plus directes, pour manipuler les chaînes.

L'union est indivisible, ainsi sont les chaînes

De ce que vous n'avez pas forcément appris à l'école, il y a au moins une chose qu'il vous faut apprendre : une fois qu'il a été créé, vous ne pouvez pas modifier un objet string. Même si je parle de modifier une chaîne, C# ne dispose d'aucune opération qui modifie l'objet string lui-même. Il existe toutes sortes d'opérations pour modifier la chaîne avec laquelle vous travaillez, mais c'est toujours avec un nouvel objet que la chaîne modifiée est retournée.

Par exemple, l'opération "Il s'appelle" + "Hector" ne modifie aucune de ces deux chaînes, mais en produit une troisième : "Il s'appelle Hector". L'une des conséquences de ce principe est que vous n'avez pas à vous inquiéter que quelqu'un modifie une chaîne "derrière votre dos".

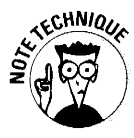
Voyez cet exemple simple :

```
// ModifyString - les méthodes fournies par la classe
//                String ne modifient pas l'objet
//                lui-même (s.ToUpper() ne modifie pas s,
//                mais retourne une nouvelle chaîne
//                qui a été convertie)
using System;
namespace Example
{
  class Class1
  {
    public static void Main(string[] args)
    {
      // crée un objet student
      Student s1 = new Student();
      s1.sName = "Jenny";
      // crée maintenant un nouvel objet avec le même nom
      Student s2 = new Student();
      s2.sName = s1.sName;
      // "changer" le nom de l'objet s1 ne change pas
      // l'objet lui-même, parce que ToUpper() retourne
      // une nouvelle chaîne sans modifier l'original
      s2.sName = s1.sName.ToUpper();
      Console.WriteLine("s1 - {0}, s2 - {1}",
                        s1.sName,
                        s2.sName);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // Student - nous avons besoin d'une classe contenant une chaîne
    class Student
    {
      public String sName;
    }
  }
}
```

Les objets `Student s1` et `s2` sont définis de telle manière que leur membre donnée `sName` pointe vers la même chaîne. L'appel à la méthode `ToUpper()` convertit la chaîne `s1.sName` pour la mettre entièrement en majuscules. Normalement, cela devrait poser un problème, car `s1` et `s2` pointent tous deux vers le même objet, mais `ToUpper()` ne modifie pas `sName` : elle crée une nouvelle chaîne en majuscules.

La sortie de ce programme est fort simple :

```
s1 - Jenny, s2 - JENNY
Appuyez sur Entrée pour terminer...
```



L'invariabilité des chaînes est également importante pour les constantes de type `string`. Une chaîne comme "ceci est une chaîne" est une forme de constante de type `string`, tout comme 1 est une constante de type `int`. De la même manière que je ne jette pas mes chemises après usage pour réduire le volume de ma garde-robe, un compilateur peut choisir de combiner tous les accès à la même constante "ceci est une chaîne". Le principe de réutilisation d'une constante de type chaîne permet de réduire la taille d'un programme, mais il serait impossible si un objet de type `string` pouvait être modifié.

Égalité pour toutes les chaînes : la méthode `Compare()`

De nombreuses opérations traitent une chaîne comme un objet unique. Par exemple, la méthode `Compare()` compare deux chaînes comme si elles étaient des nombres :

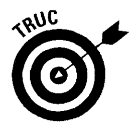
- Si la chaîne de gauche est supérieure à la chaîne de droite, `Compare()` retourne 1.
- Si la chaîne de gauche est inférieure à la chaîne de droite, `Compare()` retourne -1.
- Si les deux chaînes sont égales, `Compare()` retourne 0.

Réduit aux commentaires qui le décrivent, l'algorithme fonctionne de la façon suivante :

```
compare(string s1, string s2)
{
  // effectue une boucle sur chaque caractère des chaînes, jusqu'à
  // ce qu'un caractère d'une chaîne soit plus grand que
  // le caractère correspondant de l'autre chaîne
  foreach caractère de la chaîne la plus courte
    if (le caractère de s1 > au caractère de s2, vus comme des nombres)
      return 1
    if (le caractère de s2 < au caractère de s1)
      return -1
  // Tous les caractères correspondent, mais si la chaîne s1
```

```
    // est plus longue, alors elle est plus grande
    si s1 contient encore des caractères
      return 1
    // si s2 est plus longue, alors elle est plus grande
    si s2 contient encore des caractères
      return -1
    // si tous les caractères correspondent et si les deux chaînes
    // ont la même longueur, alors elles sont "égales"
    return 0
  }
```

Ainsi, "abcd" est plus grand que "abbd", et "abcde" est plus grand que "abcd". Vous n'aurez pas besoin tous les jours de savoir si une chaîne est plus grande qu'une autre, mais il vous arrivera d'avoir besoin de savoir si deux chaînes sont égales.



Vous aurez besoin de savoir si une chaîne est plus grande qu'une autre lorsque vous voudrez trier des chaînes.

`Compare()` retourne 0 lorsque les deux chaînes sont égales. Le programme de test suivant utilise cette caractéristique de `Compare()` pour effectuer une certaine opération quand il rencontre une ou des chaîne(s) particulière(s).

`BuildASentence` demande à l'utilisateur d'entrer des lignes de texte. Chaque ligne est concaténée avec la précédente pour former une phrase, jusqu'à ce que l'utilisateur entre les mots *EXIT*, *exit*, *QUIT*, ou *quit* :




```
// BuildASentence - le programme suivant construit
//                  des phrases en concaténant les saisies
//                  de l'utilisateur, jusqu'à ce que celui-ci entre
//                  l'un des caractères de fin -
//                  ce programme donne un exemple de la nécessité
//                  de vérifier si deux chaînes sont égales
using System;
namespace BuildASentence
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Chaque ligne que vous entrez sera"
                      + "ajoutée à une phrase, jusqu'à ce que vous"
                      + "entriez EXIT ou QUIT");
      // demande une saisie à l'utilisateur et continue à concaténer
      // jusqu'à ce que l'utilisateur entre exit ou quit
      // (commence avec une phrase vide)
      string sSentence = "";
```

```
      for(;;)
      {
        // lit la saisie suivante
        Console.WriteLine("Entrez une chaîne");
        string sLine = Console.ReadLine();
        // sort de la boucle si c'est une chaîne de fin
        if (IsTerminateString(sLine))
        {
          break;
        }
        // sinon, ajoute à la phrase la chaîne saisie
        sSentence = String.Concat(sSentence, sLine);
        // dit à l'utilisateur où il en est
        Console.WriteLine("\nVous avez entré :{0}", sSentence);
      }
      Console.WriteLine("\nPhrase complète :\n{0}", sSentence);

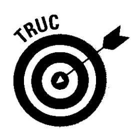
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // IsTerminateString - retourne true si la chaîne source
    // est égale à l'une des chaînes de fin
    public static bool IsTerminateString(string source)
    {
      string[] sTerms = {"EXIT",
                         "exit",
                         "QUIT",
                         "quit"};
      // compare la chaîne entrée à chacune
      // des chaînes de fin licites
      foreach(string sTerm in sTerms)
      {
        // retourne true si les deux chaînes sont égales
        if (String.Compare(source, sTerm) == 0)
        {
          return true;
        }
      }
      return false;
    }
  }
}
```

Après avoir demandé à l'utilisateur de saisir la première ligne, le programme créé une chaîne initiale vide nommée `sSentence`, puis il entre dans une boucle "infinie".



Les structures `while(true)` et `for(;;)` produisent une boucle sans fin, c'est-à-dire aussi longtemps qu'un `break` ou `return` interne n'en fait pas sortir. Les deux boucles sont équivalentes, et dans la pratique vous rencontrerez les deux.

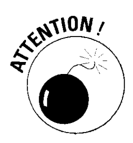
`BuildASentence` demande à l'utilisateur d'entrer une ligne de texte, qu'il lit avec la méthode `ReadLine()`. Puis il vérifie si la chaîne entrée est ou non le signal convenu pour la fin, en utilisant la chaîne créée localement, `IsTerminateString()`. Cette fonction retourne `true` si `sLine` est l'une des chaînes convenues pour la fin, et `false` dans le cas contraire.



Par convention, le nom d'une fonction qui teste une propriété et retourne `true` ou `false` doit commencer par `Is`. Dans notre exemple, le nom de la fonction `IsTerminateString()` signifie la question : "`sLine` est-elle une chaîne de fin ?" Bien sûr, ce n'est là qu'une convention humaine. Elle ne signifie rien pour C#.

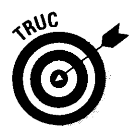
Si `sLine` n'est pas l'une des chaînes de fin, elle est concaténée avec la partie de la phrase déjà saisie, au moyen de la fonction `String.Concat()`. Le programme affiche immédiatement le résultat, afin que l'utilisateur sache où il en est.

La méthode `IsTerminateString()` définit un tableau de chaînes `sTerms`, dont chaque membre est l'une des chaînes de fin. Si la chaîne testée est égale à l'une des chaînes de ce tableau, cette méthode retourne `true`, ce qui conduit le programme à s'arrêter plus vite qu'un programmeur obligé à écrire en COBOL.



Le programme doit prendre en compte "EXIT" et "exit", car `Compare()` considère par défaut ces deux chaînes comme différentes. (À la manière dont le programme est écrit, il ne connaît que deux manières d'écrire *exit*. Une chaîne telle que "Exit" ou "eXit" ne serait pas reconnue comme chaîne de fin.)

La fonction `IsTerminateString()` effectue une boucle pour chacune des chaînes du tableau des chaînes de fin. Si `Compare()` retourne que la chaîne testée est égale à l'une des chaînes de fin du tableau, la fonction `IsTerminateString()` retourne `true`. Si aucune égalité n'a été trouvée à la fin de la boucle, la fonction `IsTerminateString()` retourne `false`.



L'itération sur un tableau est un très bon moyen de tester si une variable correspond à une valeur parmi plusieurs.

Voici un exemple de résultat du programme BuildASentence :

```
Chaque ligne que vous entrez sera ajoutée à une
phrase jusqu'à ce que vous entriez EXIT ou QUIT
Entrez une chaîne
Programmer avec C#

Vous avez entré : Programmer avec C#
Entrez une chaîne
, c'est amusant

Vous avez entré : Programmer avec C#, c'est amusant
Entrez une chaîne
 (plus ou moins)

Vous avez entré :Programmer avec C#, c'est amusant (plus ou moins)
Entrez une chaîne
EXIT

Phrase complète :
Programmer avec C#, c'est amusant (plus ou moins)
Appuyez sur Entrée pour terminer...
```

J'ai mis en gras ce qui a été saisi par l'utilisateur.

Voulez-vous comparer en majuscules ou en minuscules ?

La méthode Compare() utilisée par IsTerminateString() considère "EXIT" et "exit" comme des chaînes différentes. Mais il existe une autre version surchargée de cette fonction qui comporte un troisième argument. Celui-ci indique si la comparaison doit ou non faire la différence entre les majuscules et les minuscules. L'argument true indique d'ignorer la différence.

La version suivante de IsTerminateString() retourne true si la chaîne qui lui est passée correspond à une chaîne de fin, qu'elle soit en majuscules, en minuscules ou dans n'importe quelle combinaison des deux.

```
// IsTerminateString - retourne true si la chaîne source string is equal
// est égale à l'une des chaînes de fin
public static bool IsTerminateString(string source)
{
  //donne true si on lui passe exit ou quit, sans tenir compte
  // des majuscules et des minuscules
```

```
    return (String.Compare("exit", source, true) == 0) ||
           (String.Compare("quit", source, true) == 0);
}
```

Cette version de `IsTerminateString()` est plus simple que la précédente qui utilisait une boucle. Elle n'a pas besoin de se préoccuper des majuscules et des minuscules, et elle peut utiliser une seule instruction conditionnelle, car elle n'a maintenant que deux possibilités à prendre en compte.



Cette version de `IsTerminateString()` n'a même pas besoin d'une instruction `if`. L'expression booléenne retourne directement la valeur calculée.

Et si je veux utiliser `switch` ?

Pour tester si une chaîne est égale à une valeur particulière, vous pouvez aussi utiliser la structure `switch()`.



En général, on se sert de la structure `switch()` pour comparer une variable utilisée comme compteur à un ensemble de valeurs possibles, mais cette structure fonctionne aussi sur des chaînes.

La version suivante de `IsTerminateString()` utilise la structure `switch()` :

```
  // IsTerminateString - retourne true si la chaîne source
  // est égale à l'une des chaînes de fin
  public static bool IsTerminateString(string source)
  {
    switch(source)
    {
      case "EXIT":
      case "exit":
      case "QUIT":
      case "quit":
           return true;
    }
    return false;
  }
}
```

Cette approche fonctionne parce que vous ne comparez ici qu'un nombre limité de chaînes. Une boucle `for()` offre un moyen beaucoup plus souple de rechercher des valeurs de type chaîne. La version de `Compare()` qui ignore la distinction entre majuscules et minuscules donne au programme une plus grande souplesse.

Lire les caractères saisis

Un programme peut lire ce qui est saisi au clavier caractère par caractère, mais cette approche peut devenir problématique, car il faut se soucier des fins de ligne et autres. Une approche plus pratique consiste à lire la chaîne pour examiner ensuite les caractères qu'elle contient.

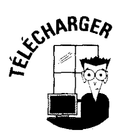
L'analyse des caractères que contient une chaîne est aussi un sujet que je n'aime pas évoquer, de crainte que les programmeurs n'abusent de cette technique. Il arrive que les programmeurs aillent un peu trop vite à sauter sur une chaîne avant qu'elle soit entièrement saisie pour en extraire ce qu'ils y trouvent. C'est particulièrement vrai des programmeurs C++, car jusqu'à l'introduction d'une classe de chaînes, c'était la seule manière dont ils pouvaient manipuler les chaînes.

En utilisant la structure `foreach` ou l'opérateur `index []`, un programme peut lire une chaîne comme si c'était un tableau de caractères.



Bien sûr, une chaîne n'est pas simplement un tableau de caractères. Si on ne peut lire une chaîne qu'un caractère à la fois, on ne peut pas l'écrire de la même manière.

L'exemple simple du programme `StringToCharAccess` montre l'utilisation de cette technique :




```
// StringToCharAccess - accède aux caractères d'une chaîne
//                      comme si la chaîne était un tableau
using System;
namespace StringToCharAccess
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // lit une chaîne saisie au clavier
      Console.WriteLine("Entrez au hasard une chaîne de caractères"
                        + "(attention : au hasard)");
      string sRandom = Console.ReadLine();
      // commence par afficher sous forme de chaîne
      Console.WriteLine("Votre saisie comme chaîne :" + sRandom);
      Console.WriteLine();
      // affiche maintenant sous forme de suite de caractères
      Console.Write("Votre saisie affichée en utilisant foreach :");
      foreach(char c in sRandom)
      {
```

```
                Console.Write(c);
            }
            Console.WriteLine(); // termine la ligne
            // put a blank line divider
            Console.WriteLine();
            // affiche maintenant sous forme de suite de caractères
            Console.Write("Votre saisie affichée en utilisant for :");
            for(int i = 0; i < sRandom.Length; i++)
            {
                Console.Write(sRandom[i]);
            }
            Console.WriteLine(); // termine la ligne
            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
        }
    }
}
```

Ce programme affiche de trois manières différentes une chaîne saisie au hasard par l'utilisateur. Il commence par l'afficher en utilisant la méthode habituelle `WriteLine(string)`, puis il l'affiche en utilisant la structure `foreach` pour en extraire chaque caractère l'un après l'autre, et enfin, il se sert de l'index d'un tableau avec `[]` pour faire la même chose.

Ce qui donne le résultat suivant :

```
Entrez au hasard une chaîne de caractères (attention : au hasard)
Stephen Davis est un beau garçon

Votre saisie comme chaîne : Stephen Davis est un beau garçon

Votre saisie affichée en utilisant foreach : Stephen Davis est un beau garçon

Votre saisie affichée en utilisant for : Stephen Davis est un beau garçon
Appuyez sur Entrée pour terminer...
```

On ne se lasse pas d'une vérité.

Dans certains cas, vous ne voudrez pas avoir un caractère non imprimable à une extrémité ou l'autre de la chaîne.



Un *caractère non imprimable* est un caractère qui n'est pas normalement affiché à l'écran : un espace, une nouvelle ligne, une tabulation, et quelques autres.

Pour épurer de ces caractères les extrémités de la chaîne, vous pouvez utiliser la méthode `Trim()` :

```
// se débarrasse des espaces à chaque extrémité d'une chaîne
sRandom = sRandom.Trim();
```



Bien que ce soit une fonction membre, `String.Trim()` retourne une nouvelle chaîne. La version précédente de la chaîne avec les caractères imprimables en surnombre est perdue et ne peut plus être utilisée.

Analyser une entrée numérique

La fonction `ReadLine()` utilisée pour lire sur la console retourne un type `string`. Un programme qui attend une entrée numérique doit convertir cette chaîne. C# offre dans la classe `Convert` l'outil de conversion dont vous avez besoin pour cela. Cette classe comporte une méthode de conversion du type `string` à tous les autres types de variable. Ainsi, le fragment de code suivant lit un nombre saisi au clavier, et le stocke dans une variable de type `int` :

```
string s = Console.ReadLine();
int n = Convert.Int32(s);
```

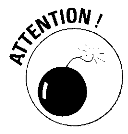
Les autres méthodes de conversion portent des noms plus évidents : `ToDouble()`, `ToFloat()`, et `ToBoolean()`.



`ToInt32()` se réfère à un entier signé de 32 bits (32 bits est la longueur d'un `int` normal). `ToInt64()` correspond à un `long` (qui fait 64 bits).

Lorsque `Convert()` rencontre un type de caractère inattendu, il peut produire un résultat inattendu. Vous devez donc être sûr du type de donnée que vous êtes en train de manier.

La fonction suivante retourne `true` si la chaîne qui lui est passée n'est constituée que de chiffres. Vous pouvez appeler cette fonction avant de convertir la chaîne en un type entier. Si une chaîne de caractère n'est constituée que de chiffres, il y a des chances que ce soit un nombre licite.



Pour une variable en virgule flottante, il serait nécessaire de prévoir la virgule, ainsi que le signe moins pour les nombres négatifs. Ne vous laissez pas surprendre.

```
// IsAllDigits - retourne true si tous les caractères de la chaîne
//               sont des chiffres
public static bool IsAllDigits(string sRaw)
{
  // commence par se débarrasser des caractères inutiles
  // à chaque extrémité ; s'il ne reste rien,
  // c'est que la chaîne n'est pas un nombre
  string s = sRaw.Trim();  // supprime les espaces aux extrémités
  if (s.Length == 0)
  {
    return false;
  }
  // effectue une boucle sur la chaîne
  for(int index = 0; index < s.Length; index++)
  {
    // si ce n'est pas un chiffre, c'est que la chaîne
    // n'est sans doute pas un nombre
    if (Char.IsDigit(s[index]) == false)
    {
      return false;
    }
  }
  // tous les caractères sont des chiffres, la chaîne doit être un nombre
  return true;
}
```

La fonction `IsAllDigits()` commence par supprimer tout caractère non imprimable aux deux extrémités de la chaîne. S'il ne reste rien, c'est que la chaîne est vide et ne peut pas être un entier. Puis, la fonction passe en boucle sur chaque caractère de la chaîne. Si l'un de ces caractères n'est pas un chiffre, la fonction retourne `false`, indiquant que la chaîne n'est sans doute pas un nombre. Si cette fonction retourne `true`, il y a les plus grandes chances que la chaîne puisse être convertie en un type entier.

L'échantillon de code suivant lit un nombre saisi au clavier, et l'affiche sur la console (pour simplifier l'exemple, j'ai omis l'utilisation de la fonction `IsAllDigits()`).



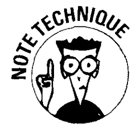

```
// IsAllDigits - démonstration de la méthode IsAllDigits
using System;
namespace Example
{
  class Class1
  {
    public static int Main(string[] args)
    {
```

```
        // lit une chaîne saisie au clavier
        Console.WriteLine("Entrez un nombre entier");
        string s = Console.ReadLine();
        // commence par vérifier si la chaîne entrée peut être un nombre
        if (!IsAllDigits(s))
        {
          Console.WriteLine("Ce n'est pas un nombre !");
        }
        else
        {
          // convertit la chaîne en un nombre entier
          int n = Int32.Parse(s);
          // affiche maintenant le double du nombre
          Console.WriteLine("2 * {0} = {1}", n, 2 * n);
        }
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
        return 0;
      }
    }
  }
```

Le programme lit sur la console une ligne saisie au clavier. Si `IsAllDigits()` retourne `false`, le programme fait des remontrances à l'utilisateur. Dans le cas contraire, le programme convertit la chaîne en nombre par l'appel `Convert.ToInt32()`. Enfin, le programme affiche le nombre ainsi que le double de celui-ci (pour bien montrer qu'il a effectivement converti la chaîne, comme il le dit).

Voici un exemple de fonctionnement de ce programme :

```
Entrez un nombre entier
1A3
Ce n'est pas un nombre !
Appuyez sur Entrée pour terminer...
```



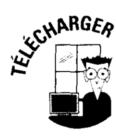
Une meilleure approche pourrait être de laisser `Convert` essayer de convertir n'importe quoi et de traiter les exceptions qui pourraient en sortir. Toutefois, il y a les plus grandes chances qu'elle ne produise aucune exception, mais retourne simplement des résultats incorrects (par exemple, 1 quand on lui propose "1A3").

Traiter une suite de chiffres

Bien souvent, un programme reçoit une suite de chiffres tapés au clavier sur une seule ligne. En utilisant la méthode `String.Split()`, vous pouvez facilement diviser cette chaîne en un certain nombre de sous-chaînes, une pour chacun des nombres dont l'ensemble est constitué, et les traiter séparément.

La fonction `Split()` divise une chaîne en un tableau de chaînes plus petites en utilisant pour cela un délimiteur. Par exemple, si vous demandez à `Split()` de diviser une chaîne en utilisant la virgule comme délimiteur, "1,2,3" produit trois chaînes : "1", "2" et "3".

Le programme suivant utilise `Split()` pour saisir une suite de nombres à additionner :




```
// ParseSequenceWithSplit - lit une série de nombres
//                  séparés par des virgules, les transforme en
//                  nombres entiers, et en affiche la somme
namespace ParseSequenceWithSplit
{
  using System;
  class Class1
  {
    public static int Main(string[] args)
    {
      // demande à l'utilisateur de saisir une série de nombres
      Console.WriteLine(
           "Entrez une série de nombres séparés par des virgules"
                     );
      // lit une ligne de texte
      string input = Console.ReadLine();
      Console.WriteLine();
      // convertit la ligne en segments
      // en utilisant la virgule ou l'espace comme séparateur
      char[] cDividers = {',', ' '};
      string[] segments = input.Split(cDividers);
      // convertit chaque segment en nombre
      int nSomme = 0;
      foreach(string s in segments)
      {
        // (saute tout segment vide)
        if (s.Length > 0)
        {
          // saute les chaînes qui ne sont pas des nombres
          if (IsAllDigits(s))
          {
```

```
            // convertit la chaîne en un entier 32 bits
            int num = Int32.Parse(s);
            Console.WriteLine("Nouveau nombre = {0}", num);
            // ajoute ce nombre à la somme
            nSum += num;
          }
        }
      }
      // affiche la somme
      Console.WriteLine("Somme = {0}", nSum);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
    // IsAllDigits - retourne true si tous les caractères
    //               sont des chiffres
    public static bool IsAllDigits(string sRaw)
    {
      // commence par se débarrasser des caractères inutiles
      // à chaque extrémité ; s'il ne reste rien
      // c'est que la chaîne n'est pas un nombre
      string s = sRaw.Trim();
      if (s.Length == 0)
      {
        return false;
      }
      // effectue une boucle sur la chaîne
      for(int index = 0; index < s.Length; index++)
      {
        // si ce n'est pas un chiffre, c'est que la chaîne
        // n'est sans doute pas un nombre
        if (Char.IsDigit(s[index]) == false)
        {
          return false;
        }
      }
      // tous les caractères sont des chiffres, c'est sans doute un nombre
      return true;
    }
  }
}
```

Le programme `ParseSequenceWithSplit` commence par lire une chaîne saisie au clavier. Il passe à la méthode `Split()` le tableau `cDividers` afin d'indiquer que la virgule et l'espace sont les caractères utilisés pour séparer deux nombres dans la chaîne.

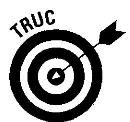
Le programme effectue une itération sur chacun des "sous-tableaux" créés par `Split()` en utilisant la structure `foreach`. Il ignore tous les sous-tableaux de longueur nulle (qui résulteraient de la présence de deux délimiteurs consécutifs). Le programme vérifie ensuite que la chaîne contient effectivement un nombre en utilisant la méthode `IsAllDigits()`. Chaque nombre valide est converti en entier puis ajouté à la variable `nSum`. Les nombres qui ne sont pas valides sont ignorés (j'ai choisi de ne pas émettre de message d'erreur).

Voici un exemple d'exécution de ce programme :

```
Entrez une série de nombres séparés par des virgules
1,2, a, 3 4

Nouveau nombre = 1
Nouveau nombre = 2
Nouveau nombre = 3
Nouveau nombre = 4
Somme = 10
Appuyez sur Entrée pour terminer...
```

Le programme parcourt cette liste, acceptant comme séparateurs la virgule, l'espace ou les deux. Il ignore le *a* et affiche le résultat 10.



Dans un programme destiné à un usage véritable, vous ne voudrez sans doute pas ignorer une donnée incorrecte sans rien signaler à l'utilisateur.

Contrôler manuellement la sortie

La maîtrise de la sortie d'un programme est un aspect très important de la manipulation des chaînes. Soyons clair : la sortie d'un programme est ce qu'en voit l'utilisateur. Quelle que soit l'élégance de sa logique interne, l'utilisateur ne sera pas bien impressionné si la sortie est plutôt piteuse.

La classe `String` offre des moyens de mettre en forme directement pour la sortie des données de type chaîne. Les sections suivantes décrivent les méthodes `Trim()`, `Pad()`, `PadRight()`, `PadLeft()`, `Substring()`, et `Concat()`.

Utiliser les méthodes `Trim()` et `Pad()`

Vous pouvez utiliser la méthode `Trim()` pour supprimer les caractères indésirables aux deux extrémités d'une chaîne. Vous allez typiquement

vous en servir pour supprimer des espaces afin d'aligner correctement les chaînes envoyées à la sortie.

Les fonctions Pad sont un autre moyen d'usage courant pour mettre en forme la sortie. Celles-ci ajoutent des caractères à l'une ou l'autre extrémité d'une chaîne pour lui donner une longueur déterminée. Par exemple, vous pourrez vouloir ajouter des espaces à l'extrémité droite ou gauche d'une chaîne pour l'aligner à droite ou gauche, ou alors ajouter des "*" ou autres caractères pour signifier quelque chose de particulier.

Le programme AlignOutput suivant utilise ces deux fonctions pour extraire et aligner une série de noms :




```
// AlignOutput - justifie à gauche et aligne un ensemble
//               de chaînes pour embellir la sortie du programme
namespace AlignOutput
{
  using System;
  class Class1
  {
    public static int Main(string[] args)
    {
      string[] names = {"Christa  ",
                        "  Sarah",
                        "Jonathan",
                        "Sam",
                        " Hildegarde  "};
      // commence par afficher les noms comme ils se présentent
      // (tout en se souvenant de la chaîne la plus longue)
      Console.WriteLine("Les noms suivants ont des "
                        + "longueurs différentes");

      foreach(string s in names)
      {
        Console.WriteLine("Ceci est le nom '{0}' initial", s);
      }
      Console.WriteLine();

      // modifie maintenant les chaînes, de manière qu'elles soient
      // justifiées à gauche et qu'elles aient toute la même longueur
      string[] sAlignedNames = TrimAndPad(names);
      // affiche enfin les chaînes modifiées,
      // justifiées et alignées
      Console.WriteLine("Voici les mêmes noms"
                      + "affichés sur la base de la même longueur");
      foreach(string s in sAlignedNames)
```

```
        {
          Console.WriteLine(
                      "Ceci est le nom '{0}' après alignement", s);
        }
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
        return 0;
      }
      // TrimAndPad - à partir d'un tableau de chaînes, supprime
      //              les espaces à chaque extrémité, puis
      //              insère les espaces nécessaires pour les aligner
      //              toutes sur la plus longue
      public static string[] TrimAndPad(string[] strings)
      {
        // copie le tableau source dans un tableau
        // que vous pourrez manipuler
        string[] stringsToAlign = new String[strings.Length];

        // commence par supprimer les espaces inutiles à chaque
        // extrémité de chaque nom
        for(int i = 0; i < stringsToAlign.Length; i++)
        {
          stringsToAlign[i] = strings[i].Trim();
        }

        // trouve maintenant la longueur de la chaîne la plus longue,
        // de façon que toutes les autres s'alignent sur elle
        int nMaxLength = 0;
        foreach(string s in stringsToAlign)
        {
          if (s.Length > nMaxLength)
          {
            nMaxLength = s.Length;
          }
        }
        // enfin, justifie toutes les chaînes sur la base
        // de la longueur de la chaîne la plus longue
        for(int i = 0; i < stringsToAlign.Length; i++)
        {
          stringsToAlign[i] =
                stringsToAlign[i].PadRight(nMaxLength + 1);
        }
        // retourne le résultat à la fonction appelante
        return stringsToAlign;
      }
    }
  }
```

`AlignOutput` définit un tableau de noms de longueur et d'alignement inégaux (on pourrait tout aussi facilement écrire un programme pour lire ces noms sur la console ou dans un fichier). La fonction `Main()` commence par afficher les noms tels qu'ils sont, puis les aligne en utilisant la méthode `TrimAndPad()` avant d'afficher à nouveau le résultat sous forme de chaînes de longueur égale avec les noms alignés à gauche :

```
Les noms suivants ont des longueurs différentes
Ceci est le nom 'Christa  ' initial
Ceci est le nom '  Sarah' initial
Ceci est le nom 'Jonathan' initial
Ceci est le nom 'Sam' initial
Ceci est le nom ' Hildegarde  ' initial

Voici les mêmes noms affichés sur la base de la même longueur
Ceci est le nom 'Christa     ' après alignement
Ceci est le nom 'Sarah       ' après alignement
Ceci est le nom 'Jonathan    ' après alignement
Ceci est le nom 'Sam         ' après alignement
Ceci est le nom 'Hildegarde  ' après alignement
```

La méthode `TrimAndPad()` commence par faire une copie du tableau de chaînes reçu. En général, une fonction qui opère sur un tableau doit retourner un nouveau tableau modifié plutôt que de modifier le tableau qui lui est passé. C'est un peu comme quand j'emprunte le pickup de mon beau-frère : il s'attend à le voir revenir dans l'état où il me l'a prêté.

`TrimAndPad()` commence par effectuer une itération sur les éléments du tableau, appelant `Trim()` sur chaque élément pour en supprimer les caractères inutiles à chaque extrémité. Puis la fonction effectue à nouveau une itération sur les éléments du tableau pour en trouver le membre le plus long. Elle effectue enfin une dernière itération, appelant `PadRight()` pour ajouter les espaces nécessaires à chaque élément, afin qu'ils aient tous la longueur du plus long.

`PadRight(10)` ajoute des espaces à l'extrémité droite d'une chaîne jusqu'à lui donner une longueur de 10 caractères. Par exemple, elle ajoute quatre espaces à l'extrémité droite d'une chaîne de six caractères.

`TrimAndPad()` retourne le tableau des chaînes allégées de leurs caractères non imprimables à droite et à gauche par `Trim()`, et mis à la bonne longueur, du bon côté, par `PadRight()`. `Main()` effectue une itération sur cette liste pour afficher l'une après l'autre toutes les chaînes.

Recoller ce que le logiciel a séparé : utiliser la concaténation

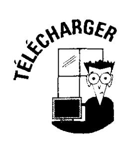
Vous serez souvent confronté à la nécessité de diviser une chaîne en plusieurs morceaux, ou d'insérer une chaîne au milieu d'une autre. Le remplacement d'un caractère par un autre est très facile à faire avec la méthode `Replace()` :

```
string s = "Danger Requins";
a.Replace(s, ' ', '!')
```

Dans cet exemple, la chaîne est convertie en "Danger!Requins".

Remplacer toutes les apparitions d'un caractère par un autre (dans ce cas, l'espace par un point d'exclamation) est particulièrement utile pour générer une chaîne contenant la virgule comme séparateur afin de la diviser ultérieurement. Toutefois, le cas le plus courant et le plus difficile est l'opération qui consiste à diviser une chaîne en plusieurs sous-ensembles, à les manipuler séparément, puis à les recombiner pour former à nouveau une seule chaîne modifiée.

Par exemple, la fonction `RemoveSpecialChars()` supprime toutes les apparitions d'un certain nombre de caractères spéciaux dans une chaîne donnée. Le programme `RemoveWhiteSpace` ci-dessous utilise cette fonction pour supprimer les caractères non imprimables (espace, tabulations et caractères de nouvelle ligne) dans une chaîne :




```
// RemoveWhiteSpace - définit une fonction RemoveSpecialChars()
//                    qui peut supprimer un caractère quelconque d'un
//                    certain ensemble d'une chaîne donnée. Utilisez
//                    cette fonction pour supprimer les caractères
//                    blancs dans une chaîne utilisée comme exemple.
namespace RemoveWhiteSpace
{
 using System;
  public class Class1
  {
    public static int Main(string[] strings)
    {
      // définit les caractères blancs
      char[] cWhiteSpace = {' ', '\n', '\t'};
      // commence par une chaîne contenant des caractères blancs
      string s = " ceci est une\nchaîne";
      Console.WriteLine("chaîne initiale :" + s);
```

```
      // affiche la chaîne sans les caractères blancs
      Console.WriteLine("après :" +
                   RemoveSpecialChars(s, cWhiteSpace));
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
    // RemoveSpecialChars - supprime de la chaîne toute
    //                      occurrence du caractère spécifié
    public static string RemoveSpecialChars(string sInput,
                                            char[] cTargets)
    {
      string sOutput = sInput;
      for(;;)
      {
        // trouve l'index du caractère, sort de la boucle
        // s'il n'en reste plus
        int nOffset = sOutput.IndexOf(cTargets);
        if (nOffset == -1)
        {
          break;
        }
        // divise la chaîne en la partie qui précède
        // le caractère et la partie qui le suit
        string sBefore = sOutput.Substring(0, nOffset);
        string sAfter  = sOutput.Substring(nOffset + 1);
        // réunit maintenant les deux sous-chaînes et le
        // caractère manquant entre les deux
        sOutput = String.Concat(sBefore, sAfter);
      }
      return sOutput;
    }
  }
}
```

C'est la fonction `RemoveSpecialChars()` qui constitue le cœur de ce programme. Elle retourne une chaîne qui est la chaîne entrée, `sInput`, mais dont tous les caractères contenus dans le tableau `cTargets` ont été supprimés. Pour mieux comprendre cette fonction, imaginez que la chaîne était "ab,cd,e", et que le tableau de caractères spéciaux à supprimer contenait simplement le caractère ','.

La fonction `RemoveSpecialChars()` entre dans une boucle dont elle ne sort qu'une fois que toutes les virgules ont été supprimées. La fonction `IndexOfAny()` retourne l'index du tableau pour la première virgule qu'elle peut trouver. Si elle retourne -1, c'est qu'aucune virgule n'a été trouvée.

À sa première invocation, `IndexOfAny()` retourne un 2 ('a' est 0, 'b' est 1, et ',' est 2). Les deux fonctions suivantes décomposent la chaîne en morceaux à l'endroit donné par l'index. `Substring(0, 2)` crée une sous-chaîne composée de deux caractères, et commençant à l'index 0 : "ab". Le deuxième appel à `Substring(3)` crée une chaîne commençant à l'index 3 et allant jusqu'à la fin de la chaîne initiale : "cd,e" (c'est le "+ 1" qui fait passer après la première virgule). C'est la fonction `Concat()` qui recolle les deux sous-chaînes pour créer "abcd,e".

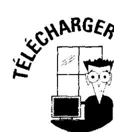
Le contrôle repasse en haut de la boucle. L'itération suivante trouve la virgule à l'index 4. La chaîne concaténée est "abcde". Comme il ne reste plus de virgule, l'index retourné par la dernière itération est -1.

Le programme `RemoveWhiteSpace` affiche une chaîne contenant plusieurs types de caractères non imprimables. Il utilise ensuite la fonction `RemoveSpecialChars()` pour enlever ces caractères non imprimables. La sortie de ce programme se présente de la façon suivante :

```
chaîne initiale : ceci est une
chaîne
après : ceciestunechaîne
Appuyez sur Entrée pour terminer...
```

Mettre `Split()` *dans le programme de concaténation*

Le programme `RemoveWhiteSpace` donne un exemple d'utilisation des méthodes `Concat()` et `IndexOf()`, mais il n'emprunte pas la voie la plus efficace. Comme d'habitude, un bref examen révèle une solution plus efficace qui utilise notre vieil ami `Split()` :




```
// RemoveSpecialChars - supprime de la chaîne toute occurrence
//                      du caractère spécifié
public static string RemoveSpecialChars(string sInput,
                                        char[] cTargets)
{
  // diviser la chaîne entrée en utilisant les caractères
  // cible comme délimiteurs
  string[] sSubStrings = sInput.Split(cTargets);
  // sOutput contiendra les informations finales de sortie
  string sOutput = "";
  // effectue une boucle sur les sous-chaînes résultant de la division
```

```
    foreach(string subString in sSubStrings)
    {
      sOutput = String.Concat(sOutput, subString);
    }
    return sOutput;
  }
```

Cette version utilise la fonction `Split()` pour diviser la chaîne entrée en un ensemble de sous-chaînes sur la base des caractères de séparation. Ceux-ci sont supprimés au passage. Ils ne font pas partie des sous-chaînes créées.

La boucle `foreach` de la deuxième partie du programme recolle les différentes sous-chaînes. La sortie du programme est la même.

Maîtriser `String.Format()`

La classe `String` offre aussi la méthode `Format()` pour mettre en forme la sortie, en particulier la sortie des nombres. Dans sa forme la plus simple, `Format()` permet d'insérer une chaîne, une variable numérique ou booléenne dans une chaîne de contrôle. Par exemple, examinez l'appel suivant :

```
String.Format("{0} fois {1} égale {2}", 2, 3, 2*3);
```

On appelle *chaîne de contrôle* le premier argument de `Format()`. Les `{n}` que vous voyez dans cette chaîne indiquent que le énième argument suivant la chaîne de contrôle doit être inséré à ce point. *Zéro* correspond au premier argument (dans ce cas, 2), *un* se réfère au suivant (3), et ainsi de suite.

Il en résulte la chaîne :

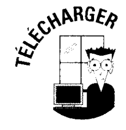
```
"2 fois 3 égale 6"
```

Sauf indication contraire, `Format()` utilise un format de sortie par défaut pour chaque type d'argument. `Format()` permet de modifier le format de sortie en mettant des modificateurs aux emplacements voulus. Le Tableau 9.1 donne une liste de certains de ces contrôles. Par exemple, `{0:E6}` dit : "Afficher les nombres en notation scientifique, en utilisant six caractères pour la mantisse."

Tableau 9.1 : Contrôles de mise en forme utilisant `String.Format()`.

Contrôle	Exemple	Résultat	Notes
C — monnaie	{0:C} avec 123,456	123,45 F	Le symbole monétaire dépend du paramétrage de localisation, de même que l'usage de la virgule ou du point comme séparateur de la partie décimale.
	{0:C} avec –123,456	(123,45 F)	
D — décimal	{0:D5} avec 123	00123	Entiers seulement.
E — exponentiel	{0:E) avec 123,45	1,2345E+02	Que l'on appelle aussi "notation scientifique".
F — fixe	{0:F2} avec 123,4567	123,45	Le nombre qui suit le F indique le nombre de chiffres après la virgule.
N — nombre	{0:N} 123456,789	123 456,79	Ajoute le séparateur de milliers (dépend du paramètre de localisation) et arrondit au centième le plus proche.
	{0:N1} 123456.789	123 456,8	Contrôle le nombre de chiffres après la virgule.
	{0:N0} 123456.789	123 457	Idem.
X — hexadécimal	{0:X}	0xFF	0xFF est égal à 255.
{0:0...}	{0:000.00} 12,3	012,30	Met un 0 s'il n'y a pas de chiffre.
{0:#...}	{0:###.##} 12,3	12,3	Impose un espace blanc si le nombre n'occupe pas l'espace spécifié. Aucun autre champ ne peut enquêter sur l'espace défini par les trois chiffres avant la virgule, et les deux après (permet de maintenir l'alignement autour de la virgule).
	{0:##0.0#} 0	0,0	Un signe # impose un espace, et un 0 impose l'affichage d'un chiffre, même si celui-ci est 0.
{0:# or 0%}	{0:#00.#%} ,1234	12,3%	Le % affiche le nombre sous forme de pourcentage (multiplie par 100 et ajoute le signe %).
	{0:#00.#%} ,0234	02,3%	

Ces contrôles de formats peuvent paraître un peu déconcertants (et je n'ai même pas parlé des contrôles détaillés de date et de format monétaire). Pour vous aider à apprivoiser ces options, le programme suivant, OutputFormatControls, vous permet d'entrer un nombre en virgule flottante suivi par une séquence de codes de contrôle. Le programme affiche alors le nombre en utilisant l'instruction Format() avec la séquence de contrôle de format spécifiée :




```
// OutputFormatControls - permet à l'utilisateur de redéfinir le format
//                        des nombres saisis en utilisant à l'exécution
//                        divers codes de contrôle de format
namespace OutputFormatControls
{
  using System;
  public class Class1
  {
    public static int Main(string[] args)
    {
      // lit les nombres saisis jusqu'à ce que
      // l'utilisateur entre une ligne blanche au lieu
      // d'un nombre
      for(;;)
      {
        // commence par lire un nombre,
        //   et se termine lorsque l'utilisateur n'entre rien
        //   qu'une ligne blanche
        Console.WriteLine("Entrez un nombre de type double");
        string sNumber = Console.ReadLine();
        if (sNumber.Length == 0)
        {
          break;
        }
        double dNumber = Double.Parse(sNumber);
        // lit maintenant les codes de contrôle, séparés
        // les uns des autres par des espaces
        Console.WriteLine("Entrez les codes de contrôle"
                          + " séparés par un espace");
        char[] separator = {' '};
        string sFormatString = Console.ReadLine();
        string[] sFormats =
                      sFormatString.Split(separator);
        // effectue une boucle sur les codes de contrôle l'un après l'autre
        foreach(string s in sFormats)
        {
          if (s.Length != 0)
          {
            // crée une commande de format complète
```

```
                // à partir des codes de contrôle entrés
                string sFormatCommand = "{0:" + s + "}";
                // affiche le nombre entré en utilisant
                // la commande de format reconstituée
                Console.Write(
                      "La commande de format {0} donne ",
                      sFormatCommand);
                try
                {
                  Console.WriteLine(sFormatCommand, dNumber);
                }
                catch(Exception)
                {
                  Console.WriteLine("<commande illégale>");
                }
                Console.WriteLine();
              }
            }
          }
          // attend confirmation de l'utilisateur
          Console.WriteLine("Appuyez sur Entrée pour terminer...");
          Console.Read();
          return 0;
        }
      }
    }
```

Le programme continue à lire dans la variable `dNumber` les nombres entrés par l'utilisateur, jusqu'à ce que celui-ci entre une ligne vide. Remarquez que le programme ne comporte aucun test pour déterminer si la valeur entrée est un nombre en virgule flottante licite. Nous supposons ici que l'utilisateur sait ce qu'est un nombre.

Le programme lit ensuite une série de codes de contrôle, séparés par des espaces. Chaque contrôle est combiné avec une chaîne "{0}" dans la variable `sFormatCommand`. Par exemple, si vous avez entré **N4**, le programme stocke la chaîne de contrôle "{0:N4}". L'instruction suivante écrit sur la console le nombre `dNumber` en utilisant la commande `sFormatCommand` ainsi construite :

```
Console.WriteLine(sFormatCommand, dNumber);
```

Dans le cas de notre N4, la commande serait :

```
Console.WriteLine("{0:N4}", dNumber);
```

Voici un exemple de sortie obtenue avec ce programme (j'ai mis en gras ce que j'ai entré) :

```
Entrez un nombre de type double
12345,6789
Entrez les codes de contrôle séparés par un espace
C E F1 N0 0000000.00000
La commande de format {0:C} donne 12 345,68 F

La commande de format {0:E} donne 1.234568E+004

La commande de format {0:F1} donne 12345,7

La commande de format {0:N0} donne 12 346

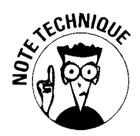
La commande de format {0:0000000.00000} donne 0012345,67890

Entrez un nombre de type double
,12345
Entrez les codes de contrôle séparés par un espace
00.0%
La commande de format {0:00.0%} donne 12,3%
Entrez un nombre de type double

Appuyez sur Entrée pour terminer...
```

Appliqué au nombre 12345,6789, la commande N0 ajoute des séparateurs de milliers aux bons endroits (c'est la partie "N") et fait disparaître tout ce qui suit la virgule décimale (c'est la partie "0"), pour afficher 12 346 (le dernier chiffre a été arrondi, et non tronqué).

De même, appliqué à 0,12345, le code de contrôle 00.0% affiche 12,3%. Le code % multiplie le nombre par 100 et ajoute le signe %. Le 00.0 indique que la sortie doit comporter au moins deux chiffres à gauche de la virgule, et seulement un à droite. Avec le même 00.0%, le nombre 0,01 est affiché comme 01,0%.



Le mystérieux `try...catch` attrape au passage toutes les erreurs qui peuvent se produire si jamais vous entrez une commande de format illicite, par exemple un "D", qui signifie décimal. Je parlerai des exceptions au Chapitre 15.

Quatrième partie

La programmation orientée objet



Dans cette partie...

La programmation orientée objet est le terme dont l'usage est accompagné de la plus grande quantité de mousse dans le monde de la programmation (il a été éclipsé pendant un an ou deux par ".com" et "e-commerce", mais vous pouvez oublier tout ça depuis le crash .com de 2001).

C++ revendique d'être un langage orienté objet. C'est ce qui le différencie de C. Java est sans aucun doute un langage orienté objet, de même qu'une centaine ou à peu près d'autres langages inventés au cours des dix dernières années. Mais que signifie *orienté objet* ? Est-ce que je l'ai ? Est-ce que je peux l'avoir ?

La quatrième partie présente les caractéristiques de C# qui en font un langage fondamentalement orienté objet.

Chapitre 10

La programmation orientée objet : qu'est-ce que c'est ?

Dans ce chapitre :

- La programmation orientée objet et le four à micro-ondes.
- Les bases de la programmation orientée objet.
- Abstraction et classification.
- Comprendre l'importance de la programmation orientée objet.

Ce chapitre apporte tout simplement la réponse à la question : "Quels sont les concepts sur lesquels repose la programmation orientée objet, et en quoi sont-ils différents de ceux que nous avons vus dans la deuxième partie de ce livre ?"

L'abstraction, concept numéro un de la programmation orientée objet

Quand je regarde un match de football (américain) à la télévision avec mon fils, il me prend souvent une envie irrésistible de nachos (chose inventée jadis par les Mexicains pour les gens qui regardent la télévision aujourd'hui). Je mets des chips dans une assiette, je les recouvre de haricots, de fromage et de beaucoup de jalapeños, et je mets le tout quelques minutes dans le four à micro-ondes.

Pour utiliser le four à micro-ondes, j'ouvre la porte, je mets l'assiette à l'intérieur, je referme la porte, et j'appuie sur quelques boutons qui se

trouvent sur la face avant. Quelques minutes plus tard, les nachos sont prêts.

Maintenant, pensez à tout ce que je ne fais pas pour utiliser le four à micro-ondes :

- Je ne change pas le câblage ni quoi que ce soit à l'intérieur du four à micro-ondes pour le faire fonctionner. Il a une interface (la face avant avec tous ses boutons et l'affichage de l'heure) qui me permet de faire tout ce dont j'ai besoin.
- Je n'ai pas à modifier le logiciel utilisé par son microprocesseur pour piloter le fonctionnement du four, même si c'est un autre plat que j'ai fait chauffer la dernière fois que je m'en suis servi.
- Je ne regarde pas sous le capot.
- Même si c'était mon métier de tout savoir sur le fonctionnement interne d'un four à micro-ondes, y compris sur son logiciel, je ne me préoccuperais pas de tout cela pour l'utiliser dans le seul but de faire chauffer mes nachos.



Ce ne sont pas là des observations profondes. On ne peut vivre avec le stress que jusqu'à une certaine limite. Pour réduire le nombre de choses dont on a à se préoccuper, on ne travaille que jusqu'à un certain niveau de détail. Dans la langue de la programmation orientée objet (OO), le niveau de détail auquel on travaille est appelé *niveau d'abstraction*. Autrement dit, pour faire chauffer mes nachos, j'ai *fait abstraction* des détails du fonctionnement interne du four à micro-ondes.

Lorsque je fais chauffer des nachos, je vois le four à micro-ondes comme une boîte. Tant que je n'utilise que l'interface du four (les boutons de la face avant), rien de ce que je fais n'est susceptible de le faire entrer dans un état instable et de le mettre hors d'usage, ou pire, de transformer mes nachos en une masse noire informe et d'y mettre le feu.

Préparer des nachos fonctionnels

Imaginez que je demande à mon fils d'écrire un algorithme décrivant la manière dont son père prépare les nachos. Quand il aura compris ce que je veux, il écrira quelque chose comme : "Ouvrir une boîte de haricots, râper du fromage, couper les jalapeños" et ainsi de suite. Une fois arrivé

au four à micro-ondes, il écrira sans doute : "Faire chauffer cinq minutes dans le four à micro-ondes."

Cette description est simple et complète, mais ce n'est pas de cette façon qu'un programmeur écrirait un programme fonctionnel pour préparer des nachos. Un programmeur vit dans un monde dépourvu d'objets tels que des fours à micro-ondes et autres appareils ménagers. Il se préoccupe généralement de diagrammes de flux, avec des milliers de chemins fonctionnels. Dans une solution fonctionnelle au problème des nachos, le contrôle passerait de mes doigts aux boutons de la face avant du four, puis à son fonctionnement interne. Très vite, le flux suivrait les chemins d'une logique complexe sur la durée pendant laquelle faire fonctionner le générateur de micro-ondes, et le moment de faire retentir la petite musique qui vous dit que c'est prêt.

Dans ce monde de programmation fonctionnelle, il n'est pas facile de penser en termes de niveaux d'abstraction. Il n'y a ici ni objets ni abstractions derrière lesquels on pourrait masquer la complexité.

Préparer des nachos orientés objet

Dans une approche orientée objet de la préparation des nachos, je commencerais par identifier les différents types d'objets intervenant dans le problème : chips, haricots, fromage et four à micro-ondes. Ensuite, j'entreprendrais la tâche de représenter ces objets dans le logiciel, sans me préoccuper des détails de leur utilisation dans le programme final.

Lorsque je fais cela, on dit que je travaille (et que je pense) au niveau des objets de base. Il me faut penser à faire un four utile, mais à ce stade je n'ai pas encore à réfléchir au processus logique de la préparation des nachos. Après tout, les concepteurs du four à micro-ondes n'ont pas pensé spécifiquement à ma manière de me préparer des nachos. Ils se sont plutôt consacrés à résoudre le problème de la conception et de la fabrication d'un four à micro-ondes utile.

Une fois que j'ai codé et testé les objets dont j'ai besoin, je peux monter au niveau d'abstraction suivant. Je peux maintenant quitter le niveau du four pour penser au niveau de la préparation des nachos. À ce stade, je peux traduire directement en code C# les instructions rédigées par mon fils.

La classification, concept numéro deux de la programmation orientée objet

La notion de classification est inséparable de la notion d'abstraction. Si je demandais à mon fils : "Qu'est-ce qu'un four à micro-ondes ?", il répondrait sans doute : "C'est un four qui..." Et si je lui demandais : "Qu'est-ce qu'un four ?", il pourrait répondre : "C'est un appareil ménager qui..." Et si je lui demandais : "Qu'est-ce qu'un appareil ménager ?", il répondrait peut-être : "Pourquoi poses-tu des questions stupides ?"

Les réponses données par mon fils dans mon exemple viennent de ce qu'il sait de notre four à micro-ondes, cas particulier du type d'objet appelé four à micro-ondes. D'autre part, mon fils considère un four à micro-ondes comme un four d'un type particulier, et un four en général comme un appareil ménager d'un type particulier.



Dans la langue de la programmation orientée objet, mon four à micro-ondes est une *instance* de la classe Four à micro-ondes. La classe Four à micro-ondes est une sous-classe de la classe Four, et la classe Four est une sous-classe de la classe Appareil ménager.

L'être humain aime classifier. Tout ce qui peuple notre monde est ordonné en taxonomies. Ce procédé nous permet de réduire le nombre de choses que nous avons à retenir. Pensez par exemple à la première fois que vous avez vu une "spacecar". La publicité la décrivait probablement comme révolutionnaire ("vous ne verrez plus jamais l'automobile de la même manière"). Et il est vrai que c'était une nouveauté, mais après tout une spacecar n'est rien d'autre qu'une voiture. En tant que telle, elle partage toutes ses propriétés (ou au moins la plupart) avec les autres voitures. Elle a un volant, des sièges, un moteur, des freins, et ainsi de suite. Je peux en conduire une sans commencer par lire le mode d'emploi.

Je n'ai pas besoin de m'encombrer la mémoire avec la liste de tout ce qu'une spacecar partage avec les autres voitures. Tout ce que j'ai à retenir est "une spacecar est une voiture qui...", et les quelques propriétés qui sont propres aux spacecars (par exemple, le prix). Mais je peux aller plus loin. La classe Voiture est une sous-classe de la classe Véhicules à roues, laquelle contient d'autres membres, comme les camions et les décapotables. Et la classe Véhicules à roues peut être une sous-classe de la classe Véhicule, qui contient les bateaux et les avions. Et ainsi de suite, aussi loin que vous voulez.

Pourquoi classifier ?

Pourquoi devrait-on classifier ? Ça a l'air de demander du travail. D'ailleurs, ça fait si longtemps qu'on utilise l'approche fonctionnelle, alors pourquoi changer maintenant ?

La conception et la fabrication d'un four à micro-ondes spécialement pour ce problème particulier peut sembler une tâche plus facile que la réalisation d'un objet four, plus générique. Supposez par exemple que je veuille fabriquer un four à micro-ondes pour faire chauffer des nachos et rien d'autre. Il ne me faudrait rien d'autre dans le panneau de commandes qu'un bouton Démarrer, car j'utilise toujours le même temps de chauffage pour mes nachos. Je pourrais me dispenser de tous les autres boutons comme Décongélation et autres. D'autre part, il n'aurait besoin de contenir rien de plus qu'une assiette. Un volume permettant de faire cuire une dinde serait ici du gaspillage.

Je peux donc me dispenser du concept de "four à micro-ondes". Je n'ai besoin que de ce qu'il fait. Puis, j'introduis dans le processus les instructions qui permettent de le faire fonctionner : "Mettre les nachos dans la boîte ; connecter le fil rouge au fil noir ; mettre le tube radar sous tension de 3 000 volts ; entendre le petit bruit qui indique le démarrage ; ne pas s'approcher trop près si on a l'intention d'avoir des enfants." Ce genre de choses.

Mais l'approche fonctionnelle a quelques inconvénients :

- **Trop compliquée :** Je ne veux pas mélanger les détails de la fabrication d'un four à micro-ondes avec ceux de la préparation des nachos. Si je ne peux pas définir les objets et les extraire de cette montagne de détails pour les utiliser de façon indépendante, je suis obligé de prendre en compte tous les détails de tous les aspects du problème en même temps.

- **Dépourvue de souplesse :** Un jour ou l'autre, je peux avoir besoin de remplacer le four à micro-ondes par un four d'un autre type. Il devrait être possible de le faire tant qu'ils ont la même interface. S'ils ne sont pas clairement délimités et développés de façon indépendante, un objet d'un certain type ne peut pas être simplement remplacé par un autre.

- **Non réutilisable :** Un four permet de faire de nombreux plats différents. Je ne veux pas avoir à créer un nouveau four pour chaque nouvelle recette. Après avoir résolu le problème une fois, je veux pouvoir réutiliser la même solution en d'autres endroits de mon programme. Et si j'ai vraiment de la chance, je pourrai même la réutiliser plus tard dans d'autres programmes.

Une interface utilisable, concept numéro trois de la programmation orientée objet

Un objet doit être capable de présenter une interface extérieure suffisante, mais aussi simple que possible. C'est un peu l'inverse du concept numéro quatre. Si l'interface de l'objet est insuffisante, les utilisateurs peuvent être amenés à en ouvrir le capot, en violation directe des lois de Dieu et de la Société (ou tout au moins en violation des lois du Texas sur la responsabilité juridique – je vous le déconseille fortement). D'un autre côté, si son interface est trop compliquée, personne n'achètera l'objet, ou en tout cas personne n'utilisera toutes ses fonctionnalités.

Les gens se plaignent régulièrement de la complexité de leurs magnétoscopes. Ils ont trop de boutons avec trop de fonctions différentes. Bien souvent, un même bouton a plusieurs fonctions différentes selon l'état de l'appareil. En plus, il n'y a pas deux modèles de magnétoscopes qui aient la même interface. Quelles qu'en soient les raisons, les magnétoscopes ont des interfaces trop compliquées et trop peu standardisées pour être utilisables par la plupart des gens.

Comparez cela avec une voiture. Il serait difficile de prétendre qu'une voiture est moins compliquée qu'un magnétoscope, mais les gens ne semblent pas avoir de difficultés à les conduire. Je vois au moins trois différences significatives entre une voiture et un magnétoscope.

Toutes les voitures présentent plus ou moins les mêmes commandes à peu près au même endroit. Par exemple (histoire vraie), ma sœur a eu une voiture (oserais-je le dire, une voiture française) dont la commande des phares était à gauche du volant, combinée avec la commande du clignotant. Il fallait pousser la manette vers le bas pour éteindre les phares, et vers le haut pour les allumer. On peut trouver que c'est une petite différence, mais je ne suis jamais arrivé à tourner à gauche de nuit avec cette voiture sans éteindre les phares.

Une voiture bien conçue n'utilise pas la même commande pour plusieurs opérations différentes selon l'état dans lequel elle se trouve. Je ne connais que très peu d'exceptions à cette règle.

Le contrôle d'accès, concept numéro quatre de la programmation orientée objet

Un four à micro-ondes doit être construit de telle sorte qu'aucune combinaison de pressions sur les boutons de la face avant ne puisse me blesser en aucune manière. Il y a certainement des combinaisons qui ne font rien, mais aucune ne doit :

- **Endommager l'appareil.** Vous devez pouvoir placer l'appareil dans une sorte d'état étrange dans lequel il ne fera rien tant que vous ne l'aurez pas ranimé, mais il doit être impossible de causer un dommage quelconque à l'appareil en utilisant les commandes de la face avant.

- **Mettre le feu à l'appareil et par conséquent à la maison.** Que l'appareil tombe en panne, c'est ennuyeux, mais qu'il prenne feu, c'est beaucoup plus grave. Nous vivons dans une société très procédurière. Il peut résulter de ce genre de choses des procès très curieux.

Toutefois, pour que ces deux règles soient respectées, vous avez aussi votre part de responsabilité : vous ne pouvez faire aucune modification à l'intérieur de l'appareil.

Presque tous les appareils ménagers, de n'importe quel niveau de complexité, notamment les fours à micro-ondes, comportent un petit sceau qui empêche le consommateur d'accéder à leurs composants internes. Si le sceau est brisé, la responsabilité du fabricant n'est plus engagée. Si je modifie les composants internes d'un four, c'est moi qui suis responsable s'il met le feu à la maison.

De même, une classe doit permettre de contrôler l'accès à ses membres. Aucune séquence d'appels aux membres d'une classe ne doit provoquer le plantage de mon programme. La classe ne peut pas le garantir si des éléments externes ont accès à ses composants et à son état interne. La classe doit pouvoir maintenir ses membres critiques inaccessibles au monde extérieur.

Comment la programmation orientée objet est-elle implémentée par C#

Dans un certain sens, ce n'est pas la bonne question. C# est un langage orienté objet : il n'implémente pas la programmation orientée objet, c'est le programmeur qui le fait. Vous pouvez écrire un programme qui ne soit pas orienté objet en C# comme dans n'importe quel autre langage, mais C# permet décrire facilement un programme orienté objet.

C# offre les fonctionnalités nécessaires à l'écriture de programmes orientés objet :

- **Le contrôle d'accès :** C# permet de contrôler la manière dont on accède à un membre. Les mots-clés C# vous permettent de déclarer certains membres ouverts au `public` alors que les membres internes (`internal`) sont protégés (`protected`) des regards extérieurs et que leurs secrets sont maintenus privés (`private`). Le Chapitre 11 vous livre les secrets du contrôle d'accès.

- **La spécialisation :** C# supporte la spécialisation à travers un mécanisme appelé *héritage de classe*. Une classe hérite des membres d'une autre classe. Par exemple, vous pouvez créer une classe `Car` comme type particulier de la classe `Vehicle`. Le Chapitre 12 est le spécialiste de la spécialisation.

- **Polymorphisme :** Cette caractéristique permet à un objet d'exécuter une opération à la manière qui lui convient. Le type `Fusée` de la classe `Véhicle` peut implémenter l'opération `Démarrage` très différemment de ce que fait le type `Voiture` de la même classe (en tout cas, j'espère que c'est toujours le cas pour ma voiture). Ces chapitres 13 et 14 ont chacun leur propre manière de décrire le polymorphisme.

Chapitre 11
Rendre une classe responsable

Dans ce chapitre :

Permettre à une classe de se protéger par le contrôle d'accès.

Permettre à un objet de s'initialiser lui-même par le constructeur.

Définir plusieurs constructeurs pour la même classe.

Construire des membres statiques ou des membres de classe.

Une classe doit être tenue pour responsable de ses actions. Tout comme un four à micro-ondes ne doit pas prendre feu si j'appuie sur le mauvais bouton, une classe ne doit pas mourir d'épouvante si je lui présente des données incorrectes.

Pour être tenue responsable de ses actions, une classe doit avoir la garantie que son état initial est correct, et pouvoir contrôler ses états suivants afin qu'ils le restent. C'est ce que permet C#.

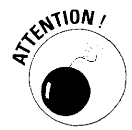
Restreindre l'accès à des membres de classe

Une classe simple définit tous ses membres comme `public`. Considérez un programme `BankAccount` qui tient à jour un membre donnée `balance` contenant le solde de chaque compte. Définir ce membre comme `public` permet à tout le monde d'y accéder.

Je ne sais pas comment est votre banque, mais la mienne est loin d'être assez confiante pour mettre à ma disposition une pile d'argent, et un

registre sur lequel il me suffirait d'inscrire ce que j'ai pris dans la pile ou ce que j'y ai ajouté. Après tout, je pourrais très bien oublier d'inscrire mes retraits dans le registre. Je ne suis plus si jeune. Ma mémoire baisse.

Le contrôle d'accès permet d'éviter les petites erreurs comme d'oublier d'inscrire un retrait ici ou là. Il permet aussi d'éviter de véritables grosses erreurs avec les retraits.



Je sais exactement ce que pensent ceux qui ont l'esprit fonctionnel : "Il suffit de définir une règle selon laquelle les autres classes ne peuvent pas accéder directement au membre `balance`." Cette approche pourrait fonctionner en théorie, mais en pratique ça ne marche pas. Les gens sont toujours plein de bonnes intentions au départ, mais ces bonnes intentions sont écrasées sous le poids de l'exigence de terminer le produit pour le livrer au client.

Un exemple public de `public BankAccount`

L'exemple suivant de classe `BankAccount` déclare toutes ses méthodes `public`, mais déclare comme `private` les deux membres donnée `nNextAccount` et `dBalance` :




```
// BankAccount - crée un compte bancaire en utilisant une variable
//               de type double pour stocker le solde du compte
//               (conserve le solde dans une variable privée
//               pour masquer son implémentation au
//               monde extérieur)
using System;
namespace DoubleBankAccount
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // crée un nouveau compte bancaire
      Console.WriteLine("Création d'un objet compte bancaire");
      BankAccount ba = new BankAccount();
      ba.InitBankAccount();
      //on peut accéder au solde par la méthode Deposit()
      // car elle a accès à tous les
      // membres donnée
      ba.Deposit(10);
      // l'accès direct à un membre donnée provoque une erreur
      // à la compilation
```

```
      Console.WriteLine("Au cas où vous arriveriez jusqu'ici"
                      + "\nCe qui suit est censé produire"
                      + "une erreur à la compilation");
      ba.dBalance += 10;
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
// BankAccount - définit une classe qui représente un compte simple
public class BankAccount
{
  private static int nNextAccountNumber = 1000;
  private int nAccountNumber;
  // conserve le solde dans une seule variable de type double
  private double dBalance;
  // Init - initialise le compte avec le prochain numéro de compte
  //        et un solde de 0
  public void InitBankAccount()
  {
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0.0;
  }
  // GetBalance - retourne le solde courant
  public double GetBalance()
  {
    return dBalance;
  }
  // AccountNumber
  public int GetAccountNumber()
  {
    return nAccountNumber;
  }
  public void SetAccountNumber(int nAccountNumber)
  {
    this.nAccountNumber = nAccountNumber;
  }
  // Deposit - tout dépôt positif est autorisé
  public void Deposit(double dAmount)
  {
    if (dAmount > 0.0)
    {
      dBalance += dAmount;
    }
  }
  // Withdraw - tout retrait est autorisé jusqu'à la valeur
  //            du solde ; retourne le montant retiré
  public double Withdraw(double dWithdrawal)
```

```
    {
      if (dBalance <= dWithdrawal)
      {
        dWithdrawal = dBalance;
      }
      dBalance -= dWithdrawal;
      return dWithdrawal;
    }
    // GetString - retourne dans une chaîne les informations sur le compte
    public string GetString()
    {
      string s = String.Format("#{0} = {1:C}",
                               GetAccountNumber(),
                               GetBalance());
      return s;
    }
  }
}
```



Souvenez-vous que dans ce code, dBalance -= dWithdrawal est la même chose que dBalance = dBalance - dWithdrawal. Les programmeurs C# ont tendance à utiliser la notation la plus concise possible.

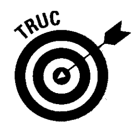
Déclarer un membre comme public le rend disponible pour n'importe quel autre code dans votre programme.

La classe BankAccount offre la méthode InitBankAccount() pour initialiser les membres de la classe, la méthode Deposit() pour traiter des dépôts, et la méthode Withdraw() pour traiter les retraits. Les méthodes Deposit() et Withdraw() fournissent même des règles rudimentaires, comme : "On ne peut pas déposer une valeur négative", et "On ne peut pas retirer plus que ce que contient le compte." Vous conviendrez certainement que ce sont de bonnes règles pour une banque. Toutefois, n'importe qui peut accéder à tout cela aussi longtemps que dBalance est accessible aux méthodes externes (dans ce contexte, externe signifie "externe à la classe, mais dans le même programme").

Avant de trop vous enthousiasmer, remarquez que ce programme ne se génère pas. Une tentative de le générer génère en fait le message d'erreur suivant :

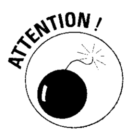
```
'DoubleBankAccount.BankAccount.dBalance' est inaccessible en raison de son
niveau de protection.
```

Je ne sais pas pourquoi il ne se contente pas de dire : "N'entrez pas, c'est privé", mais c'est essentiellement ce que ça veut dire. L'instruction `ba.dBalance += 10;` est illicite parce que `dBalance` n'est pas accessible à `Main()`. Le remplacement de cette ligne par `ba.Deposit(10)` résout le problème.



Le type d'accès par défaut est `private`. Oublier de déclarer un membre en tant que tel revient à le déclarer comme `private`, mais il vaut mieux indiquer le mot `private` pour éviter toute ambiguïté.

Allons plus loin : les autres niveaux de sécurité



Cette section suppose quelques notions sur l'héritage (Chapitre 12) et les espaces de noms (Chapitre 16). Vous pouvez l'ignorer pour le moment, mais vous saurez qu'elle est là lorsque vous en aurez besoin.

C# offre d'autres niveaux de sécurité, au-delà de `public` et `private` :

- Un membre `public` est accessible par toutes les classes du programme.
- Un membre `private` n'est accessible que par la classe dans laquelle il est déclaré.
- Un membre `protected` n'est accessible que par la classe dans laquelle il est déclaré et par toutes ses sous-classes.
- Un membre `internal` est accessible par toutes les classes du même espace de noms (essentiellement, par tout groupe de modules C# que vous aurez spécifié pour cela, c'est-à-dire tous les modules que vous aurez écrits pour le programme, mais pas ceux écrits par votre voisin de palier).
- Un membre `internal protected` est accessible par la classe dans laquelle il est déclaré et toutes ses sous-classes, ainsi que par les classes du même module.

C'est le masquage d'un membre en le déclarant `private` qui offre le maximum de sécurité. Toutefois, dans de nombreux cas, vous n'aurez pas besoin de ce niveau de sécurité. Après tout, comme les membres d'une sous-classe dépendent déjà des membres de la classe de base, `protected` offre un niveau de sécurité confortable.

Si vous déclarez chaque module comme un espace de nom différent, la déclaration d'un membre comme `internal` le rend disponible uniquement dans ce module. Mais si vous utilisez un seul espace de nom pour tous vos modules, il n'y aura guère de différence entre une déclaration `internal` ou `internal protected` et une déclaration `public`.

Pourquoi se préoccuper du contrôle d'accès ?

Déclarer les membres internes d'une classe comme `public` est une mauvaise idée, au moins pour les raisons suivantes :

- **Si tous les membres donnée sont `public`, vous ne pouvez pas savoir facilement quand et comment ils sont modifiés.** Pourquoi utiliser les méthodes `Deposit()` et `Withdraw()` pour traiter les chèques ? En fait, pourquoi avoir besoin de ces méthodes ? N'importe quelle méthode de n'importe quelle classe peut modifier ces éléments n'importe quand. Si d'autres fonctions peuvent accéder à ces membres donnée, elles le feront certainement.

  Mon programme `BankAccount` peut très bien tourner pendant une heure ou deux avant que je réalise que l'un des comptes a un solde négatif. La méthode `Withdraw()` aurait dû garantir que cela ne puisse pas arriver. De toute évidence, une autre fonction a dû accéder au solde sans passer par `Withdraw()`. Découvrir quelle fonction en est responsable et de quelle manière est un problème très difficile.

- **Exposer tous les membres donnée d'une classe rend l'interface trop compliquée.** En tant que programmeur utilisant la classe `BankAccount`, je ne veux rien savoir de ce qu'elle contient. Il me suffit de savoir qu'elle me permet de déposer et de retirer des fonds.

- **Exposer les éléments internes conduit à exporter les règles de classe.** Par exemple, la classe `BankAccount` ne permet en aucune circonstance que le solde devienne négatif. C'est une règle commerciale de la banque, qui doit être isolée dans la méthode `Withdraw()`, faute de quoi il faudra ajouter la vérification de cette règle en tout endroit du programme où le solde est modifié.

Qu'arrive-t-il lorsque la banque décide de modifier les règles pour que les "clients privilégiés" soient autorisés à avoir un solde légèrement négatif

sur une période limitée ? Il me faut maintenant rechercher dans tout le programme toutes les portions de code qui accèdent au solde afin d'y adapter en conséquence les vérifications correspondantes.

Des méthodes pour accéder à des objets

Si vous examinez plus attentivement la classe `BankAccount`, vous y verrez quelques autres méthodes. L'une d'elles, `GetString()`, retourne une version de type chaîne du compte, adaptée à l'affichage par une instruction `Console.WriteLine()`. Toutefois, l'affichage du contenu d'un objet `BankAccount` peut être difficile si ce contenu est inaccessible. D'autre part, selon la politique "Rendez à César ce qui est à César", c'est à la classe que revient le droit de décider comment elle doit être affichée.

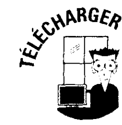
Vous remarquerez aussi une méthode, `GetBalance()`, et un ensemble de méthodes : `GetAccountNumber()` et `SetAccountNumber()`. Vous vous demandez peut-être pourquoi j'ai déclaré comme `private` un membre donnée comme `dBalance`, tout en fournissant une méthode `GetBalance()` pour en retourner la valeur. J'ai deux raisons pour cela. Tout d'abord, `GetBalance()` n'offre pas de moyen de modifier `dBalance`. Elle ne fait qu'en retourner la valeur, ce qui fait que le solde est en lecture seule. Par analogie avec une véritable banque, je peux consulter le solde de mon compte librement, mais je ne peux pas en retirer de l'argent sans passer par la procédure de retrait de la banque.

En second lieu, `GetBalance()` masque aux méthodes externes le format interne de la classe. Il est tout à fait possible que `GetBalance()` effectue de nombreux calculs concernant la lecture des reçus, des frais de gestion de compte, et tout ce que ma banque veut soustraire du solde de mon compte. Les fonctions externes n'en savent rien et n'ont rien à en faire. Naturellement, je veux savoir quels frais on m'a fait payer, mais je ne peux rien y faire, à moins de changer de banque.

Enfin, `GetBalance()` offre un mécanisme qui permet d'apporter des modifications internes à la classe sans qu'il soit nécessaire de changer les utilisateurs de `BankAccount`. Si le ministère des Finances demande à ma banque de gérer autrement ses dépôts, cela ne doit rien changer à la manière dont je peux accéder à mon compte.

Le contrôle d'accès vole à votre secours : un exemple

Le programme DoubleBankAccount suivant met en évidence un défaut potentiel dans le programme BankAccount. Le programme complet est sur le site Web, mais le listing ci-dessous ne montre que Main(), qui est la seule portion du programme comportant une différence avec BankAccount :




```
// DoubleBankAccount - crée un compte bancaire en utilisant une variable
//                     de type double pour stocker le solde du compte
//                     (conserve le solde dans une variable privée
//                     pour masquer son implémentation au
//                     monde extérieur)
namespace Test
{
  using System;
  public class Class1
  {
    public static int Main(string[] strings)
    {
      // crée un nouveau compte bancaire
      Console.WriteLine("Création d'un objet compte bancaire");
      BankAccount ba = new BankAccount();
      ba.InitBankAccount();
      // effectue un dépôt
      double dDeposit = 123.454;
      Console.WriteLine("Dépôt de {0:C}", dDeposit);
      ba.Deposit(dDeposit);
      // solde du compte
      Console.WriteLine("Compte = {0}",
                        ba.GetString());
\
      // et voilà le problème
      double dAddition = 0.002;
      Console.WriteLine("Ajout de {0:C}", dAddition);
      ba.Deposit(dAddition);
      // solde résultant
      Console.WriteLine("Compte résultant = {0}",
                        ba.GetString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
```

La fonction `Main()` crée un compte bancaire puis y dépose 123,454 F, montant qui contient un nombre décimal de centimes. `Main()` ajoute alors une petite fraction de centimes au solde, et affiche le solde résultant.

La sortie de ce programme se présente de la façon suivante :

```
Création d'un objet compte bancaire
Dépôt de 123,45 F
Compte = #1001 = 123,45 F
Ajout de 0,00 F
Compte résultant = #1001 = 123,45 F
Appuyez sur Entrée pour terminer...
```

C'est là que les utilisateurs commencent à se plaindre. Pour moi, je n'arrive pas à mettre mes chéquiers en accord avec les relevés de compte de ma banque. En fait, je suis très content si je tombe juste à 100 dollars près, mais il y a des gens qui tiennent absolument à ce que leur relevé de compte soit bon au centime près. Apparemment, il y a un bogue dans ce programme.

Le problème, bien sûr, c'est que 123,454 F apparaît comme 123,45 F. Pour éviter cela, la banque décide d'arrondir les dépôts et les retraits au centime le plus proche. Si vous déposez 123,454 F, la banque en retire les 0,4 centimes en excès. Comme elle fait la même chose lorsque la différence est en votre faveur, ça ne change rien dans la durée.

La manière la plus facile de réaliser cela consiste à convertir les comptes en `decimal` et à utiliser la méthode `RoundOff()`, comme le montre le programme `DecimalBankAccount` suivant :




```
// DecimalBankAccount - crée un compte bancaire en utilisant une
//                      variable decimal pour stocker le solde du compte
using System;
namespace DecimalBankAccount
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // crée un nouveau compte bancaire
      Console.WriteLine("Création d'un objet compte bancaire");
      BankAccount ba = new BankAccount();
      ba.InitBankAccount();
      // effectue un dépôt
      double dDeposit = 123.454;
      Console.WriteLine("Dépôt de {0:C}", dDeposit);
```

```
        ba.Deposit(dDeposit);
        // solde du compte
        Console.WriteLine("Compte = {0}",
                          ba.GetString());
        // et maintenant, ajout d'un très petit montant
        double dAddition = 0.002;
        Console.WriteLine("Ajout de {0:C}", dAddition);
        ba.Deposit(dAddition);
        // solde résultant
        Console.WriteLine("Compte résultant = {0}",
                          ba.GetString());
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
    }
}
// BankAccount - définit une classe qui représente un compte simple
public class BankAccount
{
    private static int nNextAccountNumber = 1000;
    private int nAccountNumber;
    // conserve le solde dans une seule variable de type decimal
    private decimal mBalance;
    // Init - initialise le compte avec le prochain numéro de compte
    //        et un solde de 0
    public void InitBankAccount()
    {
        nAccountNumber = ++nNextAccountNumber;
        mBalance = 0;
    }
    // GetBalance - retourne le solde courant
    public double GetBalance()
    {
        return (double)mBalance;
    }
    // AccountNumber
    public int GetAccountNumber()
    {
        return nAccountNumber;
    }
    public void SetAccountNumber(int nAccountNumber)
    {
        this.nAccountNumber = nAccountNumber;
    }
    // Deposit - tout dépôt positif est autorisé
    public void Deposit(double dAmount)
    {
        if (dAmount > 0.0)
```

```
        {
          // arrondit la variable double au centime le plus proche avant
          // d'effectuer le dépôt
          decimal mTemp = (decimal)dAmount;
          mTemp = Decimal.Round(mTemp, 2);
          mBalance += mTemp;
        }
      }
      // Withdraw - tout retrait est autorisé jusqu'à la valeur
      //            du solde ; retourne le montant retiré
      public decimal Withdraw(decimal dWithdrawal)
      {
        if (mBalance <= dWithdrawal)
        {
          dWithdrawal = mBalance;
        }
        mBalance -= dWithdrawal;
        return dWithdrawal;
      }
      // GetString - retourne dans une chaîne les informations sur le compte
      public string GetString()
      {
        string s = String.Format("#{0} = {1:C}",
                                 GetAccountNumber(),
                                 GetBalance());
        return s;
      }
    }
  }
```

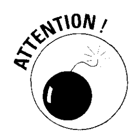
J'ai converti toutes les représentations internes en valeur `decimal`, qui est dans tous les cas un type mieux adapté que `double` au traitement de solde de compte bancaire. La méthode `Deposit()` utilise maintenant la fonction `Decimal.Round()` pour arrondir le montant des dépôts au centime le plus proche avant d'effectuer le dépôt correspondant. La sortie de ce programme est maintenant ce que nous sommes en droit d'attendre :

```
Création d'un objet compte bancaire
Dépôt de 123,45 F
Compte = #1001 = 123,45 F
Ajout de 0,00 F
Compte résultant = #1001 = 123,45 F
Appuyez sur Entrée pour terminer...
```

Et alors ?

On pourrait toujours dire que j'aurais dû écrire dès le départ le programme `BankAccount` en utilisant le type `decimal` pour les données saisies, et je serais probablement d'accord. Mais ce n'est pas si évident. Bien des applications ont été écrites en utilisant le type `double` comme moyen de stockage. Un problème s'est produit, mais la classe `BankAccount` était capable de le résoudre de façon interne sans nécessiter de modifications à l'application elle-même.

Dans ce cas, la seule fonction affectée a été `Main()`, mais les effets auraient pu s'étendre à des centaines d'autres fonctions accédant aux comptes bancaires, et ces fonctions auraient pu se trouver dans des dizaines de modules. Si la correction avait été faite à l'intérieur de la classe `BankAccount`, aucune de ces fonctions n'aurait dû être modifiée. Mais cela n'aurait pas été possible si les membres internes de la classe avaient été exposés à des fonctions externes.



Les modifications internes à une classe nécessitent quand même toujours de tester à nouveau diverses portions de code, même si celles-ci n'ont pas été modifiées.

Définir des propriétés de classe

Les méthodes `GetX()` et `SetX()` des différentes versions du programme `BankAccount` sont appelées *fonctions d'accès*, ou, plus simplement, *accesseurs*. Bien qu'en théorie elles soient synonymes de bonnes habitudes de programmation, les fonctions d'accès peuvent en pratique devenir un peu maladroites. Par exemple, le code suivant est nécessaire pour incrémenter `nAccountNumber` de 1.

```
SetAccountNumber(GetAccountNumber() + 1);
```

C# définit une structure nommée une propriété qui permet d'utiliser beaucoup plus facilement les fonctions d'accès. Le fragment de code suivant définit une propriété de lecture-écriture, `AccountNumber` :

```
public int AccountNumber
{
  get{return nAccountNumber;}
  set{nAccountNumber = value;}
}
```

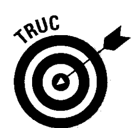
La section get est implémentée chaque fois que la propriété est lue, alors que la section set est invoquée lors de l'écriture. La propriété Balance ci-dessous est en lecture seule, car seule la section set est définie :

```
public double Balance
{
  get
  {
    return (double)mBalance;
  }
}
```

À l'utilisation, ces propriétés apparaissent de la façon suivante :

```
BankAccount ba = new BankAccount();
// stocke la propriété numéro de compte
ba.AccountNumber = 1001;
// lit les deux propriétés
Console.WriteLine("#{0} = {1:C}",
                  ba.AccountNumber, ba.Balance);
```

Les propriétés AccountNumber et Balance ressemblent beaucoup à des membres donnée publics, par leur présentation comme par leur utilisation. Toutefois, les propriétés permettent à la classe de protéger ses membres internes (Balance est une propriété en lecture seule) et de masquer leur implémentation. Remarquez que Balance effectue une conversion. Elle aurait pu aussi exécuter les calculs les plus abondants.



Par convention (ce n'est pas une obligation de C#), le nom d'une propriété commence par une lettre majuscule.

Une propriété n'est pas nécessairement dépourvue d'effet. Le compilateur C# peut optimiser un simple accesseur pour qu'il ne génère pas plus de code machine que l'accès direct à un membre donnée. C'est important, pas seulement pour l'application, mais aussi pour C# lui-même. Toute la bibliothèque C# fait un usage abondant des propriétés.

Propriétés statiques

Un membre donnée statique (de classe) peut être exposé par l'intermédiaire d'une propriété statique, comme le montre l'exemple simple suivant :

```
public class BankAccount
{
```

```
  private static int nNextAccountNumber = 1000;
  public static int NextAccountNumber
  {
    get{return nNextAccountNumber;}
  }
  // . . .
}
```

La propriété `NextAccountNumber` est accessible par la classe, car ce n'est pas la propriété d'un objet particulier :

```
// lit la propriété numéro de compte
int nValue = BankAccount.NextAccountNumber;
```

Propriétés avec effets de bord

Une opération `get` peut exécuter plus de travail que la simple extraction de la propriété associée :

```
public static int AccountNumber
{
  // extrait la propriété et prépare
  // l'extraction de la suivante
  get{return ++nNextAccountNumber;}
}
```

Cette propriété incrémente le membre statique numéro de compte avant de retourner le résultat. Toutefois, ce n'est sans doute pas une bonne idée, car l'utilisateur de la propriété n'a aucune idée de ce qui se passe en dehors de la lecture de la propriété.

Tout comme les fonctions accesseurs qu'elles imitent, les propriétés ne doivent pas changer l'état d'une classe.

Donner un bon départ à vos objets : les constructeurs

Contrôler l'accès à une classe n'est que la moitié du problème. Un objet a besoin d'un bon départ dans la vie s'il veut grandir. Une classe peut fournir une méthode d'initialisation, appelée par l'application pour faire

démarrer les choses, mais que se passe-t-il si l'application oublie d'appeler la fonction ? La classe commence avec de mauvaises initialisations, et la situation ne peut pas s'améliorer ensuite. Si vous voulez tenir la classe pour responsable de ce qu'elle fait, vous devez commencer par lui assurer un bon démarrage.

C# résout le problème en appelant la fonction d'initialisation pour vous. Par exemple :

```
MyObject mo = new MyObject();
```

En d'autres termes, non seulement cette instruction va chercher un objet dans une zone particulière de la mémoire, mais elle l'initialise en appelant la fonction d'initialisation.



Ne confondez pas les termes *classe* et *objet*. `Chien` est une classe. Mon chien `Scooter` est un objet de la classe `Chien`.

Le constructeur fourni par C#

C# se débrouille très bien pour savoir si une variable a été initialisée. Il ne vous permettra pas d'utiliser une variable non initialisée. Par exemple, le code suivant génère une erreur à la compilation :

```
public static void Main(string[] args)
{
  int n;
  double d;
  double  dCalculatedValue = n + d;
}
```

C# sait que ni `n` ni `d` n'ont reçu une valeur, et ne leur permet pas d'être utilisées dans l'expression. La compilation de ce petit programme génère les erreurs de compilation suivantes :

```
Utilisation d'une variable locale non assignée 'n'
Utilisation d'une variable locale non assignée 'd'
```

Par comparaison, C# offre un constructeur par défaut qui initialise le contenu d'un objet à 0 pour une variable intrinsèque, à `false` pour une

variable booléenne, et à `null` pour une référence d'objet. Voyez l'exemple de programme suivant :

```
using System;
namespace DecimalBankAccount
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // commence par créer un objet
      MyObject localObject =  new MyObject();
      Console.WriteLine("localObject.n est {0}", localObject.n);
      if (localObject.nextObject == null)
      {
        Console.WriteLine("localObject.nextObject est null");
      }
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
  public class MyObject
  {
    internal int n;
    internal MyObject nextObject;
  }
}
```

Ce programme définit une classe `MyObject`, qui contient une variable `n` de type `int`, et une référence à un objet, `nextObject`. La fonction `Main()` crée un objet `MyObject`, et affiche le contenu initial de `n` et de `nextObject`.

Ce programme produit la sortie suivante :

```
localObject.n est 0
localObject.nextObject est null
Appuyez sur Entrée pour terminer...
```

Lorsque l'objet est créé, C# exécute un petit morceau de code pour l'initialiser, ainsi que ses membres. Livrés à eux-mêmes, les membres donnée `localObject.n` et `nextObject` ne contiendraient que des valeurs aléatoires, sans signification.

Le code qui initialise les objets lorsqu'ils sont créés s'appelle le constructeur.

Le constructeur par défaut

C# garantit qu'un objet commence sa vie dans un état connu : rien que des zéros. Toutefois, pour de nombreuses classes (sans doute la plupart des classes), ce n'est pas un état valide. Considérez la classe BankAccount que nous avons déjà vue dans ce chapitre :

```
public class BankAccount
{
  int nAccountNumber;
  double dBalance;
  // . . .autres membres
}
```

Bien qu'un solde initial de 0 soit acceptable, un numéro de compte égal à 0 n'est certainement pas un numéro de compte valide.

La classe BankAccount contient la méthode InitBankAccount() pour initialiser l'objet. Toutefois, cette solution fait peser une responsabilité trop lourde sur l'application elle-même. Si l'application n'invoque pas la fonction InitBankAccount(), les méthodes de compte bancaire ne fonctionneront sans doute pas, sans que ce soit de leur faute. Il est préférable qu'une classe ne dépende pas de fonctions externes pour mettre ses objet dans un état valide.

En réponse à ce problème, la classe peut fournir une fonction spéciale qui sera automatiquement appelée par C# lors de la création de l'objet : le *constructeur de classe*. Celui-ci aurait pu être nommé Init(), Start(), ou Create(), pourvu que ce nom vous plaise, mais le constructeur porte le nom de la classe. Aussi, le constructeur de la classe BankAccount se présente de la façon suivante :

```
public int Main(string[] args)
{
  BankAccount ba = new BankAccount();
}
public class BankAccount
{
  // les numéros de compte commencent à 1000 et augmentent
  // séquentiellement à partir de là
  static int nNextAccountNumber = 1000;
  // met à jour le numéro de compte et le solde pour chaque objet
  int nAccountNumber;
  double dBalance;
```

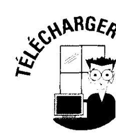
```
  // constructeur BankAccount
  public BankAccount()
  {
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0.0;
  }
  // . . . autres membres. . .
}
```

Le contenu du constructeur `BankAccount` est le même que celui de la méthode originale `Init...()`. Toutefois, une méthode n'est ni déclarée ni utilisée de la même manière :

- Le constructeur porte le même nom que la classe.
- Le constructeur n'a pas de type retourné, même pas `void`.
- `Main()` n'a pas besoin d'invoquer une fonction supplémentaire pour initialiser l'objet lorsqu'il est créé.

Construisons quelque chose

Essayez donc un de ces constructeurs. Voyez le programme `DemonstrateDefaultConstructor` suivant :




```
// DemonstrateDefaultConstructor - montre le fonctionnement
//                         des constructeurs par défaut ; crée une classe
//                         avec un constructeur, puis exécute
//                         quelques scénarios
using System;
namespace DemonstrateDefaultConstructor
{
  // MyObject - crée une classe avec un constructeur bruyant
  //            et un objet interne
  public class MyObject
  {
    // ce membre est une propriété de la classe
    static MyOtherObject staticObj = new MyOtherObject();
    // ce membre est une propriété de l'objet
    MyOtherObject dynamicObj;
    public MyObject()
    {
      Console.WriteLine("Démarrage du constructeur MyObject");
      dynamicObj = new MyOtherObject();
      Console.WriteLine("Fin du constructeur MyObject");
```

```
    }
  }
  // MyOtherObject - cette classe a aussi un constructeur bruyant
  //                mais pas de membres internes
  public class MyOtherObject
  {
    public MyOtherObject()
    {
      Console.WriteLine("Construction de MyOtherObject en cours");
    }
  }
  public class Class1
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Démarrage de Main()");
      // crée un objet
      MyObject localObject =  new MyObject();
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

L'exécution de ce programme génère la sortie suivante :

```
Démarrage de Main()
Construction de MyOtherObject en cours
Démarrage du constructeur MyObject
Construction de MyOtherObject en cours
Fin du constructeur MyObject
Appuyez sur Entrée pour terminer...
```

Reconstruisons ce qui vient de se produire :

1. Le programme démarre, et `Main()` affiche le message initial.

2. `Main()` crée un *localObject*, de type `MyObject`.

3. `MyObject` contient un membre statique, `staticObj`, de la classe `MyOtherObject`. Tous les membres donnée statiques sont créés avant que le premier `MyObject` soit construit. Dans ce cas, C# remplit `staticObj` avec un `MyOtherObject` nouvellement créé, avant de passer le contrôle au constructeur `MyObject`. Cette étape correspond au second message.

4. Le constructeur de `MyObject` reçoit le contrôle. Il affiche son premier message : `Démarrage du constructeur MyObject.`

5. Le constructeur `MyObject` crée un objet de la classe `MyOtherObject` en utilisant l'opérateur `new`, et affiche le deuxième message du constructeur `MyOtherObject`

6. Le contrôle revient au constructeur `MyObject`, qui retourne à `Main()`.

7. Mission accomplie !

Exécuter le constructeur à partir du débogueur

Pour avoir encore un peu plus de mérite, exécutez maintenant le même programme à partir du débogueur :

1. **Générez à nouveau le programme : sélectionnez Générer/Générer.**

2. **Avant de commencer à exécuter le programme à partir du débogueur, définissez un point d'arrêt à l'appel** `Console.WriteLine()` **dans le constructeur** `MyOtherObject`.



   Pour définir un point d'arrêt, cliquez dans la barre grise verticale qui constitue le bord gauche de la fenêtre de code, en regard de la ligne pour laquelle vous voulez définir un point d'arrêt.

   La Figure 11.1 montre l'affichage avec le point d'arrêt.

3. **Au lieu de sélectionner Déboguer/Démarrer, sélectionnez Déboguer/ Pas à pas détaillé (ou, mieux encore, appuyez sur la touche F11).**

   Vos fenêtres doivent s'agiter un peu pendant quelques secondes, puis l'appel `Console.WriteLine()` doit apparaître sur fond jaune.

4. **Appuyez à nouveau sur la touche F11.**

   Votre affichage doit maintenant ressembler à ce que montre la Figure 11.2.

5. **Sélectionnez Déboguer/Démarrer ou appuyez sur F5, et le programme s'exécute jusqu'au point d'arrêt dans** `MyOtherCbject`, **comme le montre la ligne en surbrillance dans la Figure 11.3.**

6. **Appuyez encore deux fois sur la touche F11, et vous êtes de nouveau au début du constructeur** `MyObject`, **comme le montre la Figure 11.4.**

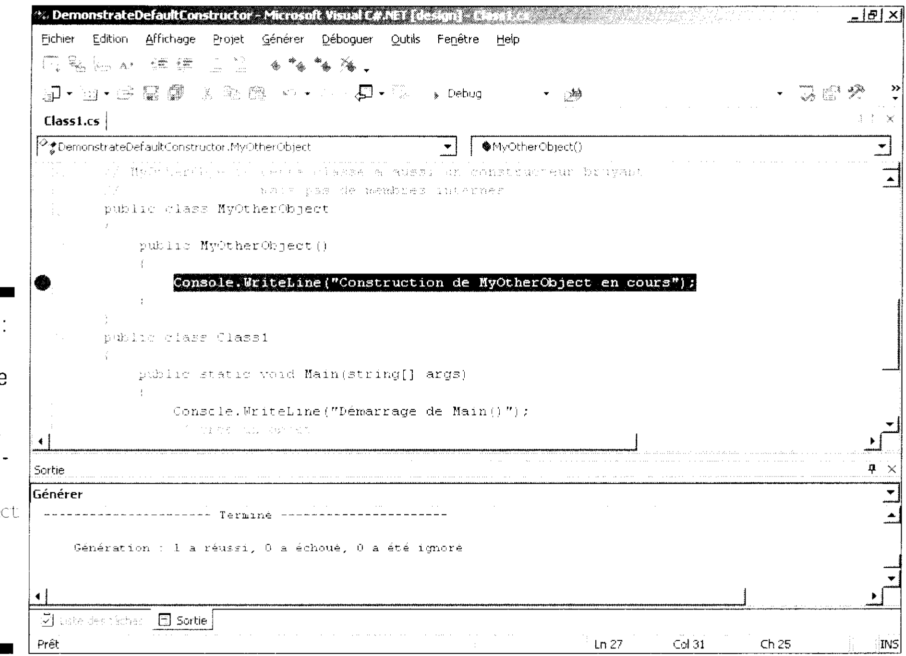


Figure 11.1 : La ligne en surbrillance sur fond rouge dans le constructeur `MyOtherObject` indique la présence d'un point d'arrêt.

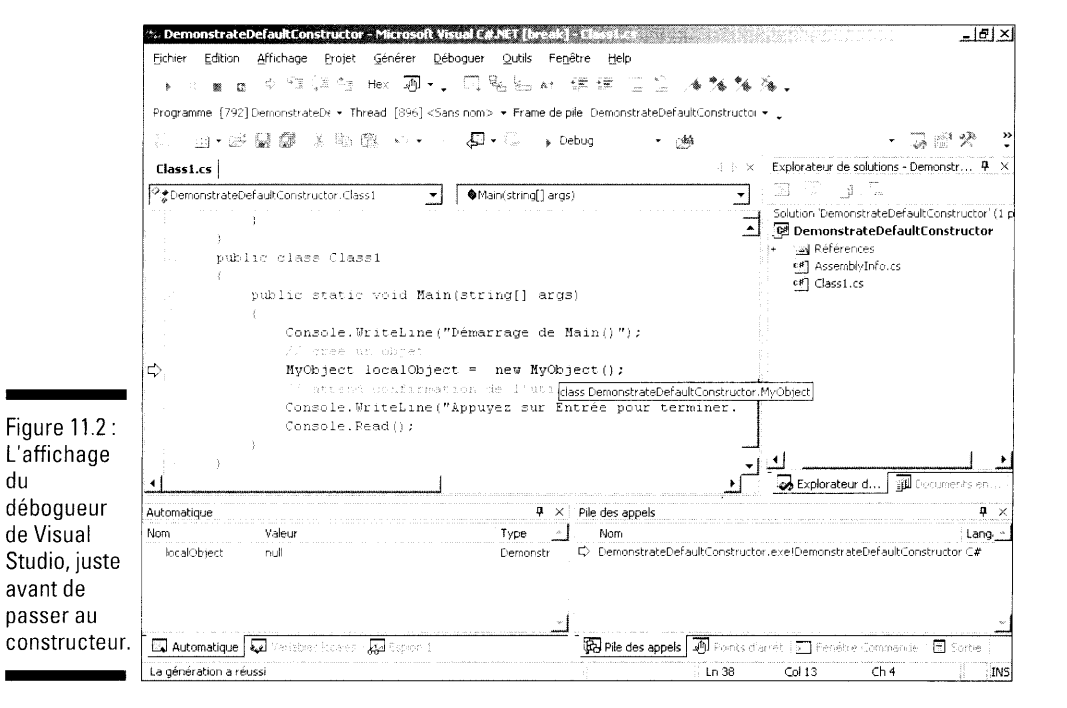


Figure 11.2 : L'affichage du débogueur de Visual Studio, juste avant de passer au constructeur.

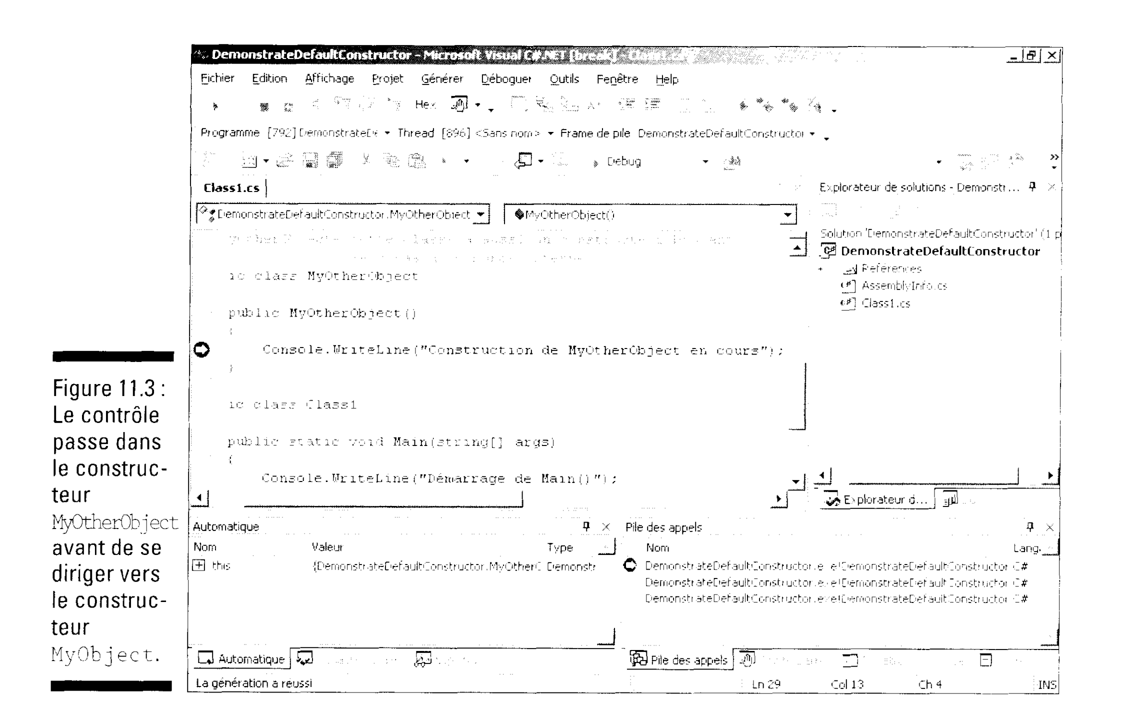


Figure 11.3 : Le contrôle passe dans le constructeur MyOtherObject avant de se diriger vers le constructeur MyObject.

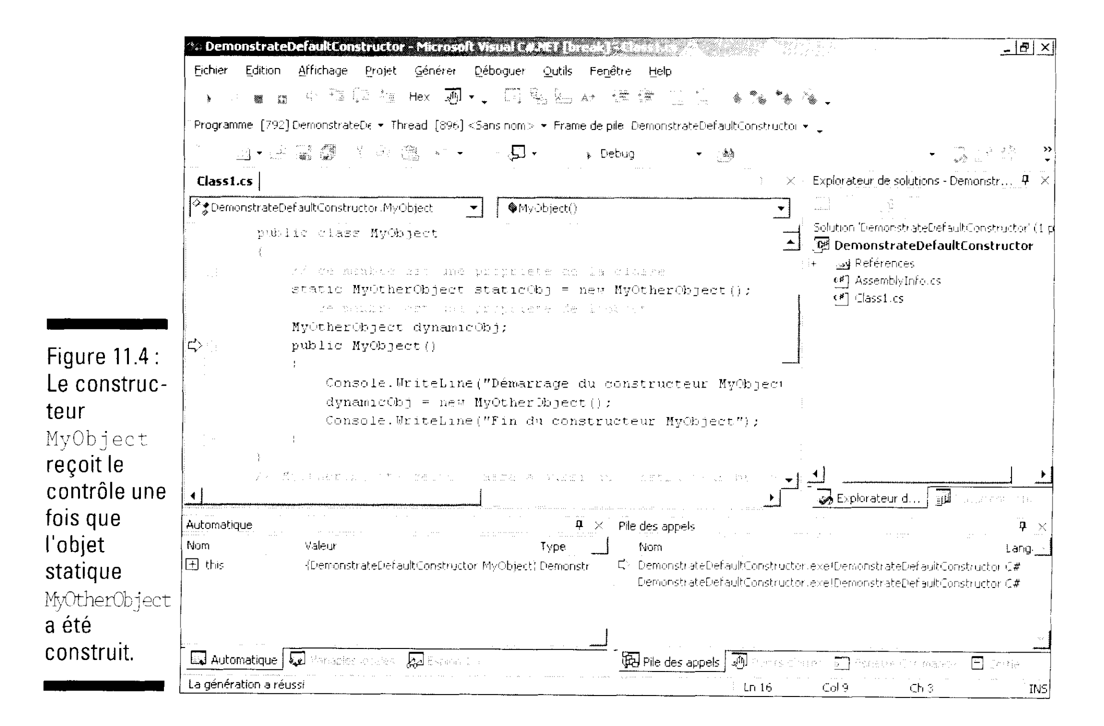


Figure 11.4 : Le constructeur MyObject reçoit le contrôle une fois que l'objet statique MyOtherObject a été construit.

7. **Continuez d'appuyer sur la touche F11 pour avancer dans le programme.**



N'oubliez pas de continuer après la commande Console.Read(). Vous devrez appuyer sur Entrée dans la fenêtre du programme avant de pouvoir continuer à avancer pas à pas dans la fenêtre du débogueur Visual Studio.

Initialiser un objet directement : le constructeur par défaut

Vous pourriez croire que presque n'importe quelle classe peut avoir un constructeur par défaut d'un certain type, et dans une certaine mesure, vous avez raison. Toutefois, C# vous permet d'initialiser directement un membre donnée en utilisant une instruction d'initialisation.

Ainsi, j'aurais pu écrire la classe BankAccount de la façon suivante :

```
public class BankAccount
{
  // les numéros de compte commencent à 1000 et augmentent
  // séquentiellement à partir de là
  static int nNextAccountNumber = 1000;
  // met à jour le numéro de compte et le solde pour chaque objet
  int nAccountNumber = ++nNextAccountNumber;
  double dBalance = 0.0;
  // . . . autres membres. . .
}
```

nAccountNumber et dBalance se voient assigner une valeur dans leur déclaration, ce qui a le même effet qu'un constructeur.

Soyons très clair sur ce qui va se passer exactement. Vous pensez peut-être que cette instruction assigne directement 0.0 à dBalance. Mais dBalance n'existe qu'en tant que partie d'un objet. Aussi, l'assignation n'est-elle pas exécutée avant qu'un objet BankAccount soit créé. En fait, cette assignation est exécutée chaque fois qu'un tel objet est créé.

C# récolte toutes les instructions d'initialisation qui apparaissent dans les déclarations de la classe, et les réunit dans un constructeur initial.

Les instructions d'initialisation sont exécutées dans l'ordre où elles se présentent dans les déclarations de la classe. Si C# rencontre des initialisations et un constructeur, les initialisations sont exécutées avant le corps du constructeur.

Voyons comment se fait la construction avec des initialisations

Déplacez maintenant l'appel `new MyOtherObject()` du constructeur `MyObject` à la déclaration elle-même, comme ci-dessous, puis exécutez à nouveau le programme :

```
public class MyObject
{
  // ce membre est une propriété de la classe
  static MyOtherObject staticObj = new MyOtherObject();
  // ce membre est une propriété de l'objet
  MyOtherObject dynamicObj = new MyOtherObject();
  public MyObject()
  {
    Console.WriteLine("Démarrage du constructeur MyObject");
    Console.WriteLine("Fin du constructeur MyObject");
  }
}
```

Le programme modifié donne la sortie suivante :

```
Démarrage de Main()
Construction de MyOtherObject en cours
Construction de MyOtherObject en cours
Démarrage du constructeur MyObject
Fin du constructeur MyObject
Appuyez sur Entrée pour terminer...
```



Vous trouverez le programme complet sur le site Web, sous le nom remarquable de `DemonstrateConstructorWithInitializer`.

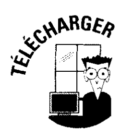
Surcharger le constructeur

On peut surcharger un constructeur, tout comme n'importe quelle autre méthode.



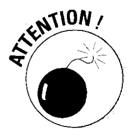
Surcharger une fonction signifie définir deux fonctions portant le même nom, mais ayant des arguments différents. Pour en savoir plus, voyez le Chapitre 7.

Imaginez que vous vouliez offrir deux manières de créer un `BankAccount` : une avec un solde à zéro, comme le mien la plupart du temps, et une autre avec une valeur initiale :




```
// BankAccountWithMultipleConstructors -
//                    fournit à notre compte bancaire
//                    un certain nombre de constructeurs,
//                    un pour chaque occasion
using System;
namespace BankAccountWithMultipleConstructors
{
  using System;
  public class Class1
  {
    public static int Main(string[] args)
    {
      // crée un compte bancaire avec des valeurs initiales valides
      BankAccount ba1 = new BankAccount();
      Console.WriteLine(ba1.GetString());
      BankAccount ba2 = new BankAccount(100);
      Console.WriteLine(ba2.GetString());
      BankAccount ba3 = new BankAccount(1234, 200);
      Console.WriteLine(ba3.GetString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
  // BankAccount - simule un simple compte bancaire
  public class BankAccount
  {
    // les numéros de compte commencent à 1000 et augmentent
    // séquentiellement à partir de là
    static int nNextAccountNumber = 1000;
```

```
    // tient à jour le numéro de compte et le solde
    int nAccountNumber;
    double dBalance;
    // fournit une série de constructeurs selon les besoins
    public BankAccount()
    {
      nAccountNumber = ++nNextAccountNumber;
      dBalance = 0.0;
    }
    public BankAccount(double dInitialBalance)
    {
      // reprend une partie du code du constructeur par défaut
      nAccountNumber = ++nNextAccountNumber;
      // et maintenant, le code propre à ce constructeur
      // commence avec le solde initial, à condition qu'il soit positif
      if (dInitialBalance < 0)
      {
        dInitialBalance = 0;
      }
      dBalance = dInitialBalance;
    }
    public BankAccount(int nInitialAccountNumber,
                       double dInitialBalance)
    {
      // ignore les numéros de compte négatifs
      if (nInitialAccountNumber <= 0)
      {
        nInitialAccountNumber = ++nNextAccountNumber;
      }
      nAccountNumber = nInitialAccountNumber;
      // commence avec le solde initial, à condition qu'il soit positif
      if (dInitialBalance < 0)
      {
        dInitialBalance = 0;
      }
      dBalance = dInitialBalance;
    }
    public string GetString()
    {
      return String.Format("#{0} = {1:N}",
                           nAccountNumber, dBalance);
    }
  }
}
```



C# ne vous offre plus de constructeur par défaut si vous définissez le vôtre, quel que soit son type.

Cette version du programme `BankAccountWithMultipleConstructors` comporte trois constructeurs :

- Le premier constructeur assigne un numéro de compte, et définit un solde égal à 0.
- Le deuxième constructeur assigne un numéro de compte, mais initialise le solde du compte avec une valeur positive. Les valeurs de solde négatives sont ignorées.
- Le troisième constructeur permet à l'utilisateur de spécifier un numéro de compte positif et un solde positif.

En utilisant chacun de ces trois constructeurs, `Main()` crée un compte bancaire différent, et affiche les objets créés. La sortie de l'exécution de ce programme se présente de la façon suivante :

```
#1001 = 0.00
#1002 = 100.00
#1234 = 200.00
Appuyez sur Entrée pour terminer...
```



Dans le monde réel, une classe effectuerait beaucoup plus de tests sur les paramètres d'entrée donnés au constructeur, pour vérifier leur validité.

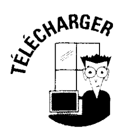
Ce sont les mêmes règles qui s'appliquent aux fonctions vous permettant de différencier les constructeurs. Le premier objet à être construit par `Main()`, `ba1`, est créé sans argument et est donc orienté vers le constructeur par défaut pour y recevoir le numéro de compte par défaut et un solde égal à zéro. Le deuxième compte, `ba2`, est envoyé au constructeur `BankAccount(double)` pour y recevoir le numéro de compte suivant, mais il est créé avec un solde initial de 100. Le troisième, `ba3`, reçoit un traitement complet, `BankAccount(int, double)`, avec son propre numéro de compte et un solde initial.

Éviter les duplications entre les constructeurs

Tout comme un scénario de série télévisée, les trois constructeurs de `BankAccount` comportent une proportion significative de duplications. Comme on peut l'imaginer, la situation serait bien pire dans des classes du monde réel qui pourraient avoir de nombreux constructeurs, et surtout bien plus de données à initialiser. De plus, les tests à effectuer sur les données saisies peuvent avoir une plus grande importance dans une classe

du monde réel que sur une page Web. La duplication de règles commerciales est à la fois fastidieuse et source d'erreurs. Les vérifications peuvent facilement se trouver en désaccord. Par exemple, du fait d'une simple erreur de codage, deux constructeurs peuvent appliquer au solde des règles différentes. De telles erreurs sont très difficiles à retrouver.

Vous préféreriez peut-être qu'un constructeur en appelle un autre, mais les constructeurs ne sont pas des fonctions : on ne peut pas les appeler. Toutefois, vous pouvez créer une alternative sous la forme d'une fonction qui effectue la véritable construction, et lui passer le contrôle, comme le montre le programme BankAccountConstructorsAndFunction ci-dessous :




```
// BankAccountContructorsAndFunction -
//                    fournit à notre compte bancaire
//                    un certain nombre de constructeurs,
//                    un pour chaque occasion
using System;
namespace BankAccountContructorsAndFunction
{
  using System;
  public class Class1
  {
    public static int Main(string[] args)
    {
      // crée un compte bancaire avec des valeurs initiales valides
      BankAccount ba1 = new BankAccount();
      Console.WriteLine(ba1.GetString());
      BankAccount ba2 = new BankAccount(100);
      Console.WriteLine(ba2.GetString());
      BankAccount ba3 = new BankAccount(1234, 200);
      Console.WriteLine(ba3.GetString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
  // BankAccount - simule un simple compte bancaire
  public class BankAccount
  {
    // les numéros de compte commencent à 1000 et augmentent
    // séquentiellement à partir de là
    static int nNextAccountNumber = 1000;
    // tient à jour le numéro de compte et le solde
    int nAccountNumber;
    double dBalance;
    // place tout le véritable code d'initialisation
```

```
    // dans une fonction conventionnelle séparée
    public BankAccount()
    {
      Init(++nAccountNumber, 0.0);
    }
    public BankAccount(double dInitialBalance)
    {
      Init(++nAccountNumber, dInitialBalance);
    }
    // c'est le constructeur le plus spécifique qui fait tout
    // le véritable travail
    public BankAccount(int nInitialAccountNumber,
                       double dInitialBalance)
    {
      Init(nInitialAccountNumber, dInitialBalance);
    }
    private void Init(int nInitialAccountNumber,
                      double dInitialBalance)
    {
      nAccountNumber = nInitialAccountNumber;
      // commence avec le solde initial, à condition qu'il soit positif
      if (dInitialBalance < 0)
      {
        dInitialBalance = 0;
      }
      dBalance = dInitialBalance;
    }
    public string GetString()
    {
      return String.Format("#{0} = {1:N}",
                           nAccountNumber, dBalance);
    }
  }
}
```

Dans cet exemple, c'est la méthode `Init()` qui fait le véritable travail de construction. Toutefois, cette approche n'est pas absolument irréprochable pour plusieurs raisons, dont l'appel d'une méthode d'un objet avant que celui-ci ait été entièrement construit n'est pas la moindre. C'est une chose très dangereuse.

Heureusement, ce n'est pas nécessaire. Un constructeur peut se référer à un autre avec une variante de l'utilisation du mot-clé `this` :

```
// BankAccountContructorsAndThis -
//                  fournit à notre compte bancaire
//                  un certain nombre de constructeurs,
```

```
//                    un pour chaque occasion
using System;
namespace BankAccountContructorsAndThis
{
  using System;
  public class Class1
  {
    public static int Main(string[] args)
    {
      // crée un compte bancaire avec des valeurs initiales valides
      BankAccount ba1 = new BankAccount();
      Console.WriteLine(ba1.GetString());
      BankAccount ba2 = new BankAccount(100);
      Console.WriteLine(ba2.GetString());
      BankAccount ba3 = new BankAccount(1234, 200);
      Console.WriteLine(ba3.GetString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
  // BankAccount - simule un simple compte bancaire
  public class BankAccount
  {
    // les numéros de compte commencent à 1000 et augmentent
    // séquentiellement à partir de là
    static int nNextAccountNumber = 1000;
    // tient à jour le numéro de compte et le solde
    int nAccountNumber;
    double dBalance;
    // invoque le constructeur spécifique en fournissant
    // des valeurs par défaut pour les arguments manquants
    public BankAccount() : this(0, 0) {}
    public BankAccount(double dInitialBalance) :
                            this(0, dInitialBalance) {}
    // c'est le constructeur le plus spécifique qui fait tout
    // le véritable travail
    public BankAccount(int nInitialAccountNumber,
                       double dInitialBalance)
    {
      // ignore les numéros de compte négatifs ; un numéro de compte nul
      // indique que nous devons utiliser le prochain numéro disponible
      if (nInitialAccountNumber <= 0)
      {
        nInitialAccountNumber = ++nNextAccountNumber;
      }
      nAccountNumber = nInitialAccountNumber;
```

```
        // commence avec le solde initial, à condition qu'il soit positif
        if (dInitialBalance < 0)
        {
          dInitialBalance = 0;
        }
        dBalance = dInitialBalance;
      }
      public string GetString()
      {
        return String.Format("#{0} = {1:N}",
                             nAccountNumber, dBalance);
      }
    }
  }
```

Cette version de BankAccount contient les trois constructeurs que nous avons vus dans la version précédente, mais au lieu de répéter les mêmes tests dans chaque constructeur, les deux constructeurs les plus simples invoquent le troisième (le plus souple), qui fournit des valeurs par défaut pour les arguments manquants.

La création d'un objet en utilisant le constructeur par défaut invoque le constructeur BankAccount() :

```
BankAccount ba1 = new BankAccount();
```

Le constructeur BankAccount() donne immédiatement le contrôle au constructeur BankAccount(int, double), en lui passant les valeurs par défaut 0 et 0.0 :

```
public BankAccount() : this(0, 0) {}
```

Le tout-puissant troisième constructeur a été modifié pour rechercher un numéro de compte nul et le remplacer par un numéro valide.

Le contrôle est restitué au constructeur par défaut une fois que le constructeur invoqué a terminé son travail. Dans ce cas, le corps du constructeur par défaut est vide.

La création d'un compte bancaire avec un solde mais un numéro de compte par défaut passe par le chemin suivant :

```
public BankAccount(double d) : this(0, d) {}
```

Être avare de ses objets

On ne peut pas construire un objet sans un constructeur correspondant. Si vous définissez votre propre constructeur, C# retire le sien. En combinant ces deux aspects, vous pouvez créer une classe qui ne peut être instanciée que localement.

Par exemple, seule une méthode définie dans le même espace de nom que `BankAccount` peut créer un objet `BankAccount` avec le constructeur déclaré comme `internal` (pour en savoir plus sur les espaces de nom, reportez-vous au Chapitre 16) :

```
// BankAccount - simule un simple compte bancaire
public class BankAccount
{
  // les numéros de compte commencent à 1000 et augmentent
  // séquentiellement à partir de là
  static int nNextAccountNumber = 1000;
  // tient à jour le numéro de compte et le solde
  int nAccountNumber;
  double dBalance;
  // invoque le constructeur spécifique en fournissant
  // des valeurs par défaut pour les arguments manquants
  internal BankAccount()
  {
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0;
  }
  public string GetString()
  {
    return String.Format("#{0} = {1:N}",
                         nAccountNumber, dBalance);
  }
}
```

Chapitre 12
Acceptez-vous l'héritage ?

Dans ce chapitre :

- Définir un nouveau type de classe, plus fondamental.
- Faire la différence entre "EST_UN" et "A_UN".
- Changer la classe d'un objet.
- Construire des membres statiques, ou de classe.
- Inclure des constructeurs dans une hiérarchie d'héritage.
- Invoquer spécifiquement le constructeur de la classe de base.

La programmation orientée objet repose sur trois principes : la possibilité de contrôler l'accès aux objets (l'encapsulation), la possibilité d'hériter d'autres classes, et la possibilité de répondre de façon appropriée (le polymorphisme).

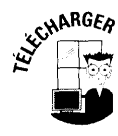
L'héritage est une notion ordinaire. Je suis un être humain, sauf à l'instant où je sors du sommeil. J'hérite de certaines propriétés de la classe `Humain`, comme ma capacité de dialoguer (plus ou moins), et ma dépendance à l'égard de l'air, de la nourriture et de boissons contenant beaucoup de caféine. La classe `Humain` hérite sa dépendance à l'égard de l'air, de l'eau et de la nourriture de la classe `Mammifère`, qui, elle-même, hérite de la classe `Animal`.

La capacité de transmettre des propriétés à un "héritier" est un aspect très puissant de la programmation orientée objet. Elle permet de décrire les choses d'une manière économique. Par exemple, si mon fils me demande : "Qu'est-ce que c'est un canard ?" Je peux répondre : "C'est un oiseau qui fait coin coin." En dépit de ce que vous pouvez penser, cette réponse contient une quantité considérable d'informations. Mon fils sait ce qu'est un oiseau, et il sait maintenant qu'un canard possède toutes les propriétés qu'il connaît des oiseaux, plus la propriété supplémentaire "faire coin coin".

Les langages orientés objet expriment cette relation d'héritage en permettant à une classe d'hériter d'une autre. C'est cette caractéristique qui permet aux langages orientés objet de produire des modèles plus proches du monde réel que les langages qui ne disposent pas du principe de l'héritage.

Hériter d'une classe

Dans l'exemple `InheritanceExample` suivant, la classe `SubClass` hérite de la classe `BaseClass` :




```
// InheritanceExample - offre la démonstration
//                       la plus simple de l'héritage
using System;
namespace InheritanceExample
{
  public class BaseClass
  {
    public int nDataMember;
    public void SomeMethod()
    {
      Console.WriteLine("SomeMethod()");
    }
  }
  public class SubClass : BaseClass
  {
    public void SomeOtherMethod()
    {
      Console.WriteLine("SomeOtherMethod()");
    }
  }
  public class Test
  {
    public static int Main(string[] args)
    {
      // crée un objet de la classe de base
      Console.WriteLine("Utilisons un objet de la classe de base :");
      BaseClass bc = new BaseClass();
      bc.nDataMember = 1;
      bc.SomeMethod();
      // créons maintenant un élément d'une sous-classe
      Console.WriteLine("Utilisons un objet d'une sous-classe :");
      SubClass sc = new SubClass();
      sc.nDataMember = 2;
      sc.SomeMethod();
```

```
            sc.SomeOtherMethod();
            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
            return 0;
        }
    }
}
```

La classe BaseClass est définie avec un membre donnée, et une simple fonction membre, SomeMethod(). L'objet bc de la classe BaseClass est créé et utilisé dans Main().

La classe SubClass hérite de la classe BaseClass en plaçant le nom de celle-ci après le signe deux-points (:) dans sa déclaration. SubClass récupère donc tous les membres de BaseClass, et peut y ajouter les siens. Main() montre que SubClass a maintenant un membre donnée, nDataMember, et une fonction membre, SomeMethod(), qui viennent rejoindre le nouveau membre de la famille, la petite méthode SomeOtherClass().

Le programme produit la sortie attendue (je suis toujours surpris quand un de mes programmes donne les résultats attendus) :

```
Utilisons un objet de la classe de base :
SomeMethod()
Utilisons un objet d'une sous-classe :
SomeMethod()
SomeOtherMethod()
Appuyez sur Entrée pour terminer...
```

Ceci est stupéfiant

Pour comprendre leur environnement, les êtres humains construisent de vastes taxonomies. Par exemple, Milou est un cas particulier de chien, qui est un cas particulier de canidé, qui est un cas particulier de mammifère, et ainsi de suite. Notre représentation du monde qui nous entoure est façonnée par cette manière de classifier les choses.

Dans un langage orienté objet comme C#, nous disons que la classe Student hérite de la classe Person. Nous disons aussi que Person est une classe de base de Student, et que Student est une sous-classe de Person. Enfin, nous disons qu'un Student **EST_UNE** Person.

Remarquez que la propriété EST_UN n'est pas réflexive : un `Student` EST_UNE `Person`, mais l'inverse n'est pas vrai. Une `Person` N'EST_PAS_UN `Student`. Un énoncé comme celui-ci se réfère toujours au cas général. Il pourrait se trouver qu'une `Person` particulière soit effectivement un `Student`, mais beaucoup de gens qui sont membres de la classe `Person` ne sont pas membres de la classe `Student`. En outre, la classe `Student` possède des propriétés qu'elle ne partage pas avec la classe `Person`. Par exemple, un `Student` a une moyenne de points d'UV, mais une `Person` ordinaire n'en a pas.

L'héritage est une propriété transitive. Par exemple, si je définis une nouvelle classe `GraduateStudent` comme une sous-classe de `Student`, alors un `GraduateStudent` est aussi une `Person`. Et il doit en être ainsi : si un `GraduateStudent` EST_UN `Student` et un `Student` EST_UNE `Person`, alors un `GraduateStudent` EST_UNE `Person`. CQFD.

À quoi me sert l'héritage ?

L'héritage a plusieurs fonctions importantes. Vous pourriez penser qu'il sert à réduire le volume de ce que vous avez à taper au clavier. Dans une certaine mesure, c'est vrai : lorsque je décris un objet de la classe `Student`, je n'ai pas besoin de répéter les propriétés d'une `Person`. Un aspect plus important, mais lié à celui-ci, est le grand mot d'ordre *réutiliser*. Les théoriciens des langages de programmation savent depuis longtemps qu'il est absurde de recommencer de zéro pour chaque nouveau projet en reconstruisant chaque fois les mêmes composants.

Comparez la situation du développement de logiciel à celle d'autres industries. Y a-t-il beaucoup de constructeurs automobile qui commencent par concevoir et fabriquer leurs propres pinces et tournevis pour construire une voiture ? Et même s'ils le faisaient, combien recommenceraient de zéro en réalisant des outils entièrement nouveaux pour chaque nouveau modèle ? Dans les autres industries, on s'est rendu compte qu'il est plus pertinent d'utiliser des vis et des écrous standards, et même des composants plus importants comme des moteurs, que de repartir de zéro chaque fois.

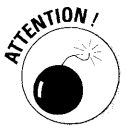
L'héritage permet de tirer le meilleur parti des composants logiciels existants. Vous pouvez adapter des classes existantes à de nouvelles applications sans leur apporter de modifications internes. C'est une nouvelle sous-classe, contenant les ajouts et les modifications nécessaires, qui hérite des propriétés d'une classe existante.

Cette capacité va de pair avec un troisième avantage de l'héritage. Imaginez que vous héritiez d'une classe existante. Un peu plus tard, vous vous apercevez que celle-ci a un bogue qu'il vous faut corriger. Si vous avez modifié la classe pour la réutiliser, vous devez rechercher manuellement le bogue, le corriger et tester le résultat, séparément, pour chaque application qui l'utilise. Si vous avez hérité de la classe sans lui faire de modifications, vous pourrez dans la plupart des cas introduire sans surprises la classe corrigée dans toutes les applications qui l'utilisent.

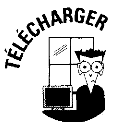
Mais le plus grand avantage de principe de l'héritage est de coller à la réalité de la vie. Les choses héritent des propriétés d'autres choses. Comme disait ma grand-mère, c'est la nature des choses.

Un exemple plus concret : hériter d'une classe BankAccount

Ma banque connaît plusieurs types de comptes bancaires. L'un d'eux, le compte rémunéré, possède des propriétés ordinaires d'un compte bancaire, plus la capacité d'accumuler des intérêts. L'exemple de programme suivant, `SimpleSavingsAccount`, réalise en C# un modèle de ces relations :



Que les plus impressionnables d'entre vous ne s'affolent pas : ce listing est un peu long, mais il est divisé en parties clairement distinctes.




```
// SimpleSavingsAccount - implémente un SavingsAccount comme forme
//                d'un BankAccount ; n'utilise pas de méthode virtuelle
using System;
namespace SimpleSavingsAccount
{
  // BankAccount - simule un compte bancaire possédant
  //               un numéro de compte (assigné à la création
  //               du compte) et un solde
  public class BankAccount
  {
    // les numéros de compte commencent à 1000 et augmentent
    // séquentiellement à partir de là
    public static int nNextAccountNumber = 1000;
    // met à jour le numéro de compte et le solde pour chaque objet
    public int nAccountNumber;
    public decimal mBalance;
    // Init - initialise le compte avec le prochain numéro de compte
    //        et le solde initial spécifié
```

```
    //        (qui est égal à zéro par défaut)
    public void InitBankAccount()
    {
      InitBankAccount(0);
    }
    public void InitBankAccount(decimal mInitialBalance)
    {
      nAccountNumber = ++nNextAccountNumber;
      mBalance = mInitialBalance;
    }
    // Balance (solde)
    public decimal Balance
    {
      get { return mBalance;}
    }
    // Deposit - tout dépôt positif est autorisé
    public void Deposit(decimal mAmount)
    {
      if (mAmount > 0)
      {
        mBalance += mAmount;
      }
    }
    // Withdraw - tout retrait est autorisé jusqu'à la valeur
    //            du solde ; retourne le montant retiré
    public decimal Withdraw(decimal mWithdrawal)
    {
      if (mBalance <= mWithdrawal)
      {
        mWithdrawal = mBalance;
      }
      mBalance -= mWithdrawal;
      return mWithdrawal;
    }
    // ToString - met le compte sous forme de chaîne
    public string ToBankAccountString()
    {
      return String.Format("{0} - {1:C}",
        nAccountNumber, mBalance);
    }
  }
  // SavingsAccount - compte bancaire qui rapporte des intérêts
  public class SavingsAccount : BankAccount
  {
    public decimal mInterestRate;
    // InitSavingsAccount - lit le taux d'intérêt, exprimé en
    //                      pourcentage (valeur comprise entre 0 et 100)
    public void InitSavingsAccount(decimal mInterestRate)
```

```
    {
      InitSavingsAccount(0, mInterestRate);
    }
    public void InitSavingsAccount(decimal mInitial,
                                   decimal mInterestRate)
    {
      InitBankAccount(mInitial);
      this.mInterestRate = mInterestRate / 100;
    }
    // AccumulateInterest - invoquée une fois par période
    public void AccumulateInterest()
    {
      mBalance = mBalance + (decimal)(mBalance * mInterestRate);
    }
    // ToString - met le compte sous forme de chaîne
    public string ToSavingsAccountString()
    {
      return String.Format("{0} ({1}%)",
        ToBankAccountString(), mInterestRate * 100);
    }
  }
  public class Class1
  {
    public static int Main(string[] args)
    {
      // crée un compte bancaire et l'affiche
      BankAccount ba = new BankAccount();
      ba.InitBankAccount(100);
      ba.Deposit(100);
      Console.WriteLine("Compte {0}", ba.ToBankAccountString());
      // et maintenant un compte rémunéré
      SavingsAccount sa = new SavingsAccount();
      sa.InitSavingsAccount(100, 12.5M);
      sa.AccumulateInterest();
      Console.WriteLine("Compte {0}", sa.ToSavingsAccountString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
}
```

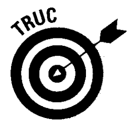
La classe `BankAccount` n'est guère différente de celles qui apparaissent dans d'autres chapitres de ce livre. Elle commence par la fonction d'initialisation surchargée `InitBankAccount()` : une pour les comptes qui sont créés avec un solde initial, une autre pour ceux qui devront se contenter de commencer de zéro.

La propriété Balance permet de lire le solde, mais sans donner la possibilité de le modifier. La méthode Deposit() accepte tout dépôt positif. La méthode Withdraw() vous permet de retirer tout ce que vous voulez dans la limite de ce que vous avez sur votre compte. ToBankAccountString() crée une chaîne qui donne la description du compte.

La classe SavingsAccount hérite de toutes ces bonnes choses de BankAccount. À cela, elle ajoute un taux d'intérêt, et la possibilité d'accumuler des intérêts à intervalle régulier.

Main() en fait le moins possible. Elle crée un BankAccount, affiche le compte, crée un SavingsAccount, ajoute une période d'intérêts, et affiche le résultat :

```
Compte 1001 - 200,00 ¤
Compte 1002 - 112,50 ¤ (12,5%)
Appuyez sur Entrée pour terminer...
```



Remarquez que la méthode InitSavingsAccount() invoque InitBankAccount(). Cela initialise les membres donnée propres au compte. La méthode InitSavingsAccount() aurait pu les initialiser directement, mais il est de meilleure pratique de permettre à BankAccount d'initialiser ses propres membres.

EST_UN par rapport à A_UN – j'ai du mal à m'y retrouver

La relation entre SavingsAccount et BankAccount n'est rien d'autre que la relation fondamentale EST_UN. Pour commencer, je vais vous montrer pourquoi, puis je vous montrerai à quoi ressemblerait une relation A_UN.

La relation EST_UN

La relation EST_UN entre SavingsAccount et BankAccount est mise en évidence par la modification suivante à Class1 dans le programme SimpleSavingsAccount de la section précédente :

```
public class Class1
{
  // DirectDeposit - effectue automatiquement le dépôt d'un chèque
  public static void DirectDeposit(BankAccount ba,
```

```
                                      decimal mPay)
    {
      ba.Deposit(mPay);
    }
    public static int Main(string[] args)
    {
      // crée un compte bancaire et l'affiche
      BankAccount ba = new BankAccount();
      ba.InitBankAccount(100);
      DirectDeposit(ba, 100);
      Console.WriteLine("Compte {0}", ba.ToBankAccountString());
      // et maintenant un compte rémunéré
      SavingsAccount sa = new SavingsAccount();
      sa.InitSavingsAccount(12.5M);
      DirectDeposit(sa, 100);
      sa.AccumulateInterest();
      Console.WriteLine("Compte {0}", sa.ToSavingsAccountString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
```

Les effets de ce programme n'ont pas changé. La seule véritable différence est que tous les dépôts sont maintenant effectués par la fonction locale `DirectDeposit()`. Les arguments de cette fonction sont le compte bancaire et le montant à déposer.

Remarquez (c'est le bon côté de la chose) que `Main()` peut passer à `DirectDeposit()` soit un compte ordinaire, soit un compte rémunéré, car un `SavingsAccount` EST_UN `BankAccount` et en reçoit par conséquent tous les droits et privilèges.

Contenir BankAccount *pour y accéder*

La classe `SavingsAccount` aurait pu accéder d'une autre manière aux membres de `BankAccount` :

```
// SavingsAccount - compte bancaire qui rapporte des intérêts
public class SavingsAccount_
{
  public BankAccount bankAccount;
  public decimal mInterestRate;
  // InitSavingsAccount - lit le taux d'intérêt, exprimé en
```

```
  //                      pourcentage (valeur comprise entre 0 et 100)
  public void InitSavingsAccount(BankAccount bankAccount,
                                 decimal mInterestRate)
  {
    this.bankAccount = bankAccount;
    this.mInterestRate = mInterestRate / 100;
  }
  // AccumulateInterest - invoquée une fois par période
  public void AccumulateInterest()
  {
    bankAccount.mBalance = bankAccount.mBalance
                  + (bankAccount.mBalance * mInterestRate);
  }
  // Deposit - tout dépôt positif est autorisé
  public void Deposit(decimal mAmount)
  {
    bankAccount.Deposit(mAmount);
  }
  // Withdraw - tout retrait est autorisé jusqu'à la valeur
  //            du solde ; retourne le montant retiré
  public double Withdraw(decimal mWithdrawal)
  {
    return bankAccount.Withdraw(mWithdrawal);
  }
}
```

Ici, la classe SavingsAccount_ contient un membre donnée bankAccount (au lieu d'en hériter de BankAccount). L'objet bankAccount contient le solde et le numéro du compte, informations nécessaires pour la gestion du compte rémunéré. Les données propres à un compte rémunéré sont contenues dans la classe SavingsAccount_.

Dans ce cas, nous disons que SavingsAccount_ A_UN BankAccount.

La relation A_UN

La relation A_UN est fondamentalement différente de la relation EST_UN. Cette différence ne semble pas mauvaise dans l'exemple de code suivant :

```
// crée un nouveau compte rémunéré
BankAccount ba = new BankAccount()
SavingsAccount_ sa = new SavingsAccount_();
sa.InitSavingsAccount(ba, 5);
// et y dépose cent euros
sa.Deposit(100);
```

```
// puis accumule des intérêts
sa.AccumulateInterest();
```

Le problème est qu'un `SavingsAccount_` ne peut pas être utilisé comme un `BankAccount`. Par exemple, le code suivant ne marche pas :

```
// DirectDeposit - effectue automatiquement le dépôt d'un chèque
void DirectDeposit(BankAccount ba, int nPay)
{
  ba.Deposit(nPay);
}
void SomeFunction()
{
  // l'exemple qui suit ne marche pas
  SavingsAccount_ sa = new SavingsAccount_();
  DirectDeposit(sa, 100);
  // . . . suite. . .
}
```

`DirectDeposit()` ne peut pas accepter un `SavingsAccount_` en lieu et place d'un `BankAccount`. C# ne peut voir aucune relation évidente entre les deux.

Quand utiliser EST_UN et quand utiliser A_UN ?

La distinction entre les relations EST_UN et A_UN est plus qu'une question de commodité logicielle. Cette relation a un corollaire dans le monde réel.

Par exemple, une Ford Explorer EST_UNE voiture (quand elle n'est pas sur le toit). Une Explorer A_UN moteur. Si un ami me dit : "Viens avec ta voiture", et que j'arrive dans une Explorer, il n'a aucun reproche à me faire (ou alors, s'il en a, ce n'est pas parce que l'Explorer n'est pas une voiture). Il pourrait me faire des reproches si j'arrivais en portant dans mes bras le moteur de mon Explorer.

La classe `Explorer` doit apporter une extension à la classe `Car`, non seulement pour donner à `Explorer` accès aux méthodes de `Car`, mais aussi pour exprimer la relation fondamentale entre les deux.

Malheureusement, un programmeur débutant peut faire hériter `Car` de `Motor`, donnant à la classe `Car` accès aux membres de `Motor`, dont `Car` a besoin pour fonctionner. Par exemple, `Car` peut hériter de la méthode `Motor.Go()`, mais cet exemple met en lumière l'un des problèmes qui résultent de cette approche. Même si les êtres humains s'expriment parfois

de façon ambiguë, faire démarrer une voiture n'est pas la même chose que faire démarrer un moteur. L'opération démarrage de la voiture dépend évidemment du démarrage du moteur, mais ce sont deux choses distinctes : il faut aussi passer la première, lâcher les freins, et ainsi de suite.

Plus encore, sans doute, faire hériter `Car` de `Motor` est une représentation erronée des choses. Une voiture n'est tout simplement pas un type particulier de moteur.



L'élégance du logiciel est un but qui se passe de justification. Non seulement elle le rend plus compréhensible, plus fiable et aisé à maintenir, mais elle réjouit le goût et facilite la digestion, entre autres.

Autres considérations

C# implémente un ensemble de caractéristiques conçues pour supporter l'héritage.

Changer de classe

Un programme peut changer la classe d'un objet. En fait, c'est une chose que vous avez déjà vue dans cet exemple. `SomeFunction()` peut passer un objet `SavingsAccount` à une méthode qui attend un objet `BankAccount`.

Vous pouvez rendre cette conversion plus explicite :

```
BankAccount ba;
SavingsAccount sa = new SavingsAccout();
                        // OK:
ba = sa;                // une conversion vers le bas implicite est admise
ba = (BankAccount)sa;   // le cast explicite est préféré
                        // Non!
sa = ba;                // la conversion vers le haut implicite est interdite
                        // ceci est correct
sa = (SavingsAccount)ba;
```

La première ligne stocke un objet `SavingsAccount` dans une variable `BankAccount`. C# effectue pour vous cette conversion. La deuxième ligne utilise l'opérateur cast pour convertir explicitement l'objet.

Les deux dernières lignes reconvertissent l'objet `BankAccount` en `SavingsAccount`.



La propriété EST_UN n'est pas réflexive. Autrement dit, même si Explorer est une voiture, une voiture n'est pas nécessairement une Explorer. De même, un `BankAccount` n'est pas nécessairement un `SavingsAccount`, et la conversion implicite n'est donc pas autorisée. La dernière ligne est admise parce que le programmeur a indiqué sa volonté de "tenter le coup".

Des casts invalides à l'exécution

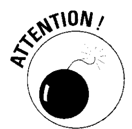
En général, le casting d'un objet de `BankAccount` à `SavingsAccount` est une opération dangereuse. Considérez l'exemple suivant :

```
public static void ProcessAmount(BankAccount bankAccount)
{
  // dépose une grosse somme sur le compte
  bankAccount.Deposit(10000.00);
  // si l'objet est un SavingsAccount,
  // recueille l'intérêt dès maintenant
  SavingsAccount SavingsAccount = (SavingsAccount)bankAccount;
  savingsAccount.AccumulateInterest();
}
public static void TestCast()
{
  SavingsAccount sa = new SavingsAccount();
  ProcessAmount(sa);
  BankAccount ba = new BankAccount();
  ProcessAmount(ba);
}
```

`ProcessAmount()` exécute un certain nombre d'opérations, dont l'invocation de la méthode `AccumulateInterest()`. Le cast de `ba` à un `SavingsAccount` est nécessaire, parce que `ba` est déclaré comme un `BankAccount`. Le programme se compile correctement, parce que toutes les conversions de type sont faites par un cast explicite.

Tout se passe bien avec le premier appel à `ProcessAmount()`, dans `Test()`. L'objet `sa` de la classe `SavingsAccount` est passé à la méthode `ProcessAmount()`. Le cast de `BankAccount` à `SavingsAccount` ne pose pas de problème parce que l'objet `ba` était de toute façon à l'origine un objet `SavingsAccount`.

Le deuxième appel à `ProcessAmount()` n'est toutefois pas aussi chanceux. Le cast à `SavingsAccount` ne peut pas être autorisé. L'objet `ba` n'a pas de méthode `AccumulateInterest()`.



Une conversion incorrecte génère une erreur à l'exécution du programme (ce qu'on appelle une erreur *run-time*). Une erreur à l'exécution est beaucoup plus difficile à identifier et corriger qu'une erreur à la compilation.

Éviter les conversions invalides en utilisant le mot-clé `is`

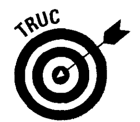
La fonction `ProcessAmount()` se porterait très bien si elle pouvait être sûre que l'objet qui lui est passé est bien un `SavingsAccount` avant d'effectuer la conversion. C'est dans ce but que C# offre le mot-clé `is`.

L'opérateur `is` admet un objet à sa gauche et un type à sa droite. Il retourne `true` si le type à l'exécution de l'objet qui est à sa gauche est compatible avec le type qui est à sa droite.

Vous pouvez modifier l'exemple précédent pour éviter l'erreur à l'exécution en utilisant l'opérateur `is` :

```
public static void ProcessAmount(BankAccount bankAccount)
{
  // dépose une grosse somme sur le compte
  bankAccount.Deposit(10000.00);
  // si l'objet est un SavingsAccount . . .
  if (bankAccount is SavingsAccount)
  {
    // ...recueille l'intérêt dès maintenant
    SavingsAccount SavingsAccount = (SavingsAccount)bankAccount;
    savingsAccount.AccumulateInterest();
  }
}
public static void TestCast()
{
  SavingsAccount sa = new SavingsAccount();
  ProcessAmount(sa);
  BankAccount ba = new BankAccount();
  ProcessAmount(ba);
}
```

L'instruction `if` supplémentaire teste l'objet `bankAccount` pour vérifier qu'il est bien de la classe `SavingsAccount`. L'opérateur `is` retourne `true` lorsque `ProcessAmount()` est appelée pour la première fois. Toutefois, lorsqu'un objet `bankAccount` lui est passé dans le deuxième appel, l'opérateur `is` retourne `false`, évitant ainsi le cast invalide. Cette version de ce programme ne génère pas d'erreur à l'exécution.



D'un côté, je vous recommande fortement de protéger tous vos casts vers le haut avec l'opérateur `is` pour éviter le risque d'une erreur à l'exécution, d'un autre côté, je vous conseille d'éviter tout cast vers le haut, si possible.

La classe `object`

Les classes suivantes sont en relation les unes avec les autres :

```
public class MyBaseClass {}
public class MySubClass : MyBaseClass {}
```

La relation entre ces deux classes permet au programmeur d'effectuer le test suivant à l'exécution :

```
public class Test
{
  public static void GenericFunction(MyBaseClass mc)
  {
    // si l'objet est vraiment une sous-classe...
    if (mc is MySubClass)
    {
      // ...alors le traite comme une sous-classe
      MySubClass msc = (MySubClass)mc;

      // . . . suite . . .
    }
  }
}
```

Dans ce cas, la fonction `GenericFunction()` différencie les sous-classes de `MyBaseClass` en utilisant le mot-clé `is`.

Comment faire la différence entre ces classes, apparemment sans lien entre elles, en utilisant le même opérateur `is` ? C# étend toutes ces classes à partir de leur classe de base commune, `object`. Autrement dit, toute classe qui n'hérite pas spécifiquement d'une autre classe hérite de la classe `object`. Ainsi, les deux déclarations suivantes sont identiques :

```
class MyClass1 : object {}
class MyClass2 {}
```

`MyClass1` et `MyClass2` ont en commun la classe de base `object`, ce qui autorise la fonction générique suivante :

```
public class Test
{
```

```
    public static void GenericFunction(object o)
    {
      if (o is MyClass1)
      {
        MyClass1 mc1 = (MyClass1)o;
        // . . .
      }
    }
  }
```

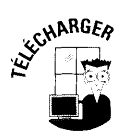
`GenericFunction()` peut être invoquée avec n'importe quel type d'objet. Le mot-clé `is` extraira des huîtres `object` toutes les perles de `MyClass1`.

L'héritage et le constructeur

Le programme `InheritanceExample` que nous avons vu plus haut dans ce chapitre repose sur ces horribles fonctions `Init...` pour initialiser les objets `BankAccount` et `SavingsAccount` en leur donnant un état valide. Équiper ces classes de constructeurs est certainement la meilleure manière de procéder, mais elle introduit une petite complication.

Invoquer le constructeur par défaut de la classe de base

Le constructeur par défaut de la classe de base est invoqué chaque fois qu'une sous-classe est construite. Le constructeur de la sous-classe invoque automatiquement le constructeur de la classe de base, comme le montre cet exemple simple :




```
// InheritingAConstructor - montre que le constructeur
//                          de la classe de base est invoqué
//                          automatiquement
using System;
namespace InheritingAConstructor
{
  public class Class1
  {
```

```
    public static int Main(string[] args)
    {
      Console.WriteLine("Création d'un objet BaseClass");
      BaseClass bc = new BaseClass();
      Console.WriteLine("\nMaintenant, création d'un objet SubClass");
      SubClass sc = new SubClass();
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
  public class BaseClass
  {
    public BaseClass()
    {
      Console.WriteLine("Construction de BaseClass");
    }
  }
  public class SubClass : BaseClass
  {
    public SubClass()
    {
      Console.WriteLine("Construction de SubClass");
    }
  }
}
```

Les constructeurs de `BaseClass` et `SubClass` ne font rien de plus qu'afficher un message sur la ligne de commande. La création de l'objet `BaseClass` invoque le constructeur par défaut de la classe `BaseClass`. La création d'un objet `SubClass` invoque le constructeur de `BaseClass` avant d'invoquer son propre constructeur.

Ce programme donne la sortie suivante :

```
Création d'un objet BaseClass
Construction de BaseClass

Maintenant, création d'un objet SubClass
Construction de BaseClass
Construction de SubClass
Appuyez sur Entrée pour terminer...
```

Une hiérarchie de classes héritées ressemble beaucoup aux différents étages d'un immeuble. Chaque classe se trouve au-dessus des classes

dont elle réalise une extension. Il y a une raison à cela : chaque classe est responsable de ce qu'elle fait. Une sous-classe ne doit pas plus être tenue pour responsable de l'initialisation des membres de la classe de base qu'une fonction extérieure quelconque. La classe `BaseClass` doit se voir donner la possibilité de construire ses membres avant que les membres de `SubClass` aient la possibilité d'y accéder.

Passer des arguments au constructeur de la classe de base : le mot-clé `base`

La sous-classe invoque le constructeur par défaut de sa classe de base, sauf indication contraire, même à partir d'un constructeur d'une sous-classe autre que le constructeur par défaut. C'est ce que montre l'exemple légèrement modifié ci-dessous :

```
using System
namespace Example
{
  public class Class1
  {
    public static int Main(string[] args)
    {
      Console.WriteLine("Invocation de SubClass()");
      SubClass sc1 = new SubClass();
      Console.WriteLine("\nInvocation de SubClass(int)");
      SubClass sc2 = new SubClass(0);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
  public class BaseClass
  {
    public BaseClass()
    {
      Console.WriteLine("Construction de BaseClass (default)");
    }
    public BaseClass(int i)
    {
      Console.WriteLine("Construction de BaseClass (int)");
    }
  }
  public class SubClass : BaseClass
```

```
  {
    public SubClass()
    {
      Console.WriteLine("Construction de SubClass (default)");
    }
    public SubClass(int i)
    {
      Console.WriteLine("Construction de SubClass (int)");
    }
  }
}
```

L'exécution de ce programme donne les résultats suivants :

```
Invocation de SubClass()
Construction de BaseClass (default)
Construction de SubClass (default)

Invocation de SubClass(int)
Construction de BaseClass (default)
Construction de SubClass (int)
Appuyez sur Entrée pour terminer...
```

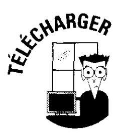
Le programme commence par créer un objet par défaut. Comme prévu, C# invoque le constructeur par défaut de `SubClass`, qui commence par passer le contrôle au constructeur par défaut de `BaseClass`. Le programme crée alors un objet, en passant un argument entier. À nouveau comme prévu, C# invoque `SubClass(int)`. Ce constructeur invoque le constructeur par défaut de `BaseClass`, comme dans l'exemple précédent, car il n'a pas de données à passer.

Un constructeur d'une sous-classe peut invoquer un constructeur particulier de la classe de base en utilisant le mot-clé `base`.



Ce procédé est très similaire à la manière dont un constructeur en invoque un autre de la même classe en utilisant le mot-clé `this`. Pour tout savoir sur les constructeurs avec `this`, voyez le Chapitre 11.

Par exemple, examinez le petit programme `InvokeBaseConstructor` :




```
// InvokeBaseConstructor - montre comment une sous-classe peut
//                         invoquer le constructeur de la classe de base
//                         de son choix en utilisant le mot-clé base
using System;
namespace InvokeBaseConstructor
```

```
{
  public class BaseClass
  {
    public BaseClass()
    {
      Console.WriteLine("Construction de BaseClass (default)");
    }
    public BaseClass(int i)
    {
      Console.WriteLine("Construction de BaseClass({0})", i);
    }
  }
  public class SubClass : BaseClass
  {
    public SubClass()
    {
      Console.WriteLine("Construction de SubClass (default)");
    }
    public SubClass(int i1, int i2) : base(i1)
    {
      Console.WriteLine("Construction de SubClass({0}, {1})",
                                            i1,  i2);
    }
  }
  public class Class1
  {
    public static int Main(string[] args)
    {
      Console.WriteLine("Invocation de SubClass()");
      SubClass sc1 = new SubClass();
      Console.WriteLine("\nInvocation de SubClass(1, 2)");
      SubClass sc2 = new SubClass(1, 2);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
}
```

Ce programme donne la sortie suivante :

```
Invocation de SubClass()
Construction de BaseClass (default)
Construction de SubClass (default)

Invocation de SubClass(1, 2)
```

```
Construction de BaseClass(1)
Construction de SubClass(1, 2)
Appuyez sur Entrée pour terminer...
```

Cette version commence de la même manière que les exemples précédents, en créant un objet SubClass par défaut, utilisant les constructeurs par défaut de BaseClass et de SubClass.

Le deuxième objet est créé avec l'expression SubClass(1, 2). C# invoque le constructeur SubClass(int, int), qui utilise le mot-clé base pour passer l'une des valeurs au constructeur BaseClass(int). On peut se douter que SubClass passe le premier argument à la classe de base pour traitement, et continue en utilisant la deuxième valeur pour elle-même.

La classe BankAccount *modifiée*

Le programme ConstructorSavingsAccount est une version modifiée du programme SimpleBankAccount. Dans cette version, le constructeur de SavingsAccount peut repasser des informations aux constructeurs de BankAccount. Seuls Main() et les constructeurs eux-mêmes apparaissent ici :




```
// ConstructorSavingsAccount - implémente un SavingsAccount
//                             comme forme d'un BankAccount ; n'utilise
//                             aucune méthode virtuelle, mais implémente
//                             correctement les constructeurs
using System;
namespace ConstructorSavingsAccount
{
  // BankAccount - simule un compte bancaire possédant
  //               un numéro de compte (assigné à la création
  //               du compte) et un solde
  public class BankAccount
  {
    // les numéros de compte commencent à 1000 et augmentent
    // séquentiellement à partir de là
    public static int nNextAccountNumber = 1000;
    // met à jour le numéro de compte et le solde pour chaque objet
    public int nAccountNumber;
    public decimal mBalance;
    // Constructeurs
    public BankAccount():this(0)
    {
    }
    public BankAccount(decimal mInitialBalance)
```

```
    {
      nAccountNumber = ++nNextAccountNumber;
      mBalance = mInitialBalance;
    }
    // . . . même chose ici . . .
  }
  // SavingsAccount - compte bancaire qui rapporte des intérêts
  public class SavingsAccount : BankAccount
  {
    public decimal mInterestRate;
    // InitSavingsAccount - lit le taux d'intérêt, exprimé en
    //                      pourcentage (valeur comprise entre 0 et 100)
    public SavingsAccount(decimal mInterestRate) : this(0, mInterestRate)
    {
    }
    public SavingsAccount(decimal mInitial,
                          decimal mInterestRate) : base(mInitial)
    {
      this.mInterestRate = mInterestRate / 100;
    }
    // . . . même chose ici . . .
  }
  public class Class1
  {
    // DirectDeposit - effectue automatiquement le dépôt d'un chèque
    public static void DirectDeposit(BankAccount ba,
                                     decimal mPay)
    {
      ba.Deposit(mPay);
    }
    public static int Main(string[] args)
    {
      // crée un compte bancaire et l'affiche
      BankAccount ba = new BankAccount(100);
      DirectDeposit(ba, 100);
      Console.WriteLine("Compte {0}", ba.ToBankAccountString());
      // et maintenant un compte rémunéré
      SavingsAccount sa = new SavingsAccount(12.5M);
      DirectDeposit(sa, 100);
      sa.AccumulateInterest();
      Console.WriteLine("Compte {0}", sa.ToSavingsAccountString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
}
```

BankAccount définit deux constructeurs : un qui admet un solde initial, et le constructeur par défaut qui ne l'admet pas. Afin d'éviter toute publication de code dans le constructeur, le constructeur par défaut invoque le constructeur de BankAccount(initial balance) en utilisant le mot-clé this.

La classe SavingsAccount fournit également deux constructeurs. Le constructeur SavingsAccount(interest rate) invoque le constructeur SavingsAccount(interest rate, initial balance) en lui passant un solde initial de 0. Ce constructeur, le plus général, passe le solde initial au constructeur BankAccount(initial balance) en utilisant le mot-clé base, comme le montre de façon graphique la Figure 12.1.

Figure 12.1 : Le cheminement de la construction d'un objet SavingsAccount, en utilisant le constructeur par défaut.

J'ai modifié Main() pour me débarrasser de ces infernales fonctions Init...(), en les remplaçant par des constructeurs. La sortie de ce programme est la même :

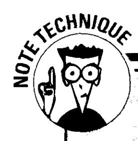
```
Compte 1001 - 200,00 €
Compte 1002 - 112,50 € (12,5%)
Appuyez sur Entrée pour terminer...
```

Le destructeur

C# offre aussi une méthode qui fait l'inverse du constructeur, appelée le destructeur. Le destructeur porte le nom de la classe, précédé par un tilde (~). Par exemple, la méthode ~BaseClass est le destructeur de BaseClass.

C# invoque le destructeur lorsqu'il n'utilise plus l'objet. Le destructeur par défaut est le seul qui peut être créé, car le destructeur ne peut être invoqué directement. En outre, le destructeur est toujours virtuel.

Dans le cas d'une succession d'héritages de classes, les destructeurs sont invoqués dans l'ordre inverse des constructeurs. Autrement dit, le destructeur de la sous-classe est invoqué avant le destructeur de la classe de base.



Le ramasse-miettes et le destructeur C#

La méthode du destructeur est beaucoup moins utile en C# que dans d'autres langages orientés objet, comme C++, car C# possède de ce l'on appelle une destruction non déterministe.

La mémoire allouée à un objet est supprimée du tas lorsque le programme exécute la commande `new`. Ce bloc de mémoire reste réservé aussi longtemps que les références valides à celui-ci restent actives.

Une zone de mémoire est dite "inaccessible" lorsque la dernière référence à celle-ci passe hors de portée. Autrement dit, personne ne peut plus accéder à cette zone de mémoire quand plus rien n'y fait référence.

C# ne fait rien de particulier lorsqu'une zone de mémoire devient inaccessible. Une tâche de faible priorité est exécutée à l'arrière-plan, recherchant les zones de mémoire inaccessibles. Ce qu'on appelle le *ramasse-miettes* s'exécute à un faible niveau de priorité afin d'éviter de diminuer les performances du programme. Le ramasse-miettes restitue au tas les zones de mémoire inaccessibles qu'il trouve.

En temps normal, le ramasse-miettes opère en silence à l'arrière-plan. Il ne prend le contrôle du programme qu'à de brefs moments, lorsque le tas est sur le point d'être à court de mémoire.

Le destructeur de C# est non déterministe parce qu'il ne peut pas être invoqué avant que l'objet ait été récupéré par le ramasse-miettes, ce qui peut se produire longtemps après qu'il a cessé d'être utilisé. En fait, si le programme se termine avant que l'objet soit trouvé par le ramasse-miettes et retourné au tas, le destructeur n'est pas invoqué du tout.

Au bout du compte, l'effet qui en résulte est qu'un programmeur C# ne peut pas se reposer sur le destructeur pour opérer automatiquement comme dans un langage comme C++.

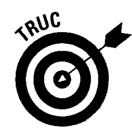
Chapitre 13

Quel est donc ce polymorphisme ?

Dans ce chapitre :

- L'embarras du choix : masquer ou écraser une méthode de la classe de base.
- Construire des classes abstraites : parlez-vous sérieusement ?
- Déclarer comme abstraites une méthode et la classe qui la contient.
- Faire commencer une nouvelle hiérarchie au-dessus d'une hiérarchie existante.
- Empêcher qu'une classe puisse être transformée en sous-classe.

L'héritage permet à une classe "d'adopter" les membres d'une autre. Ainsi, je peux créer une classe `SavingsAccount` qui hérite de membres donnée comme `mBalance` et de méthodes comme `Deposit()` de la classe de base `BankAccount`. C'est très joli, mais cette définition de l'héritage ne suffit pas à représenter convenablement ce qui se passe dans le monde réel.



Si vous avez besoin de vous rafraîchir la mémoire sur l'héritage des classes, relisez le Chapitre 14.

Un four à micro-ondes est un type de four, non pas parce qu'il a l'air d'un four, mais parce qu'il remplit les mêmes fonctions qu'un four. Un four à micro-ondes remplit aussi des fonctions supplémentaires, mais au moins, il remplit les fonctions de base d'un four – et le plus important, c'est qu'il fait chauffer mes nachos quand je dis "`StartCooking`". Le processus interne que doit mettre en œuvre le four pour remplir sa mission ne m'intéresse pas, pas plus que le type de four dont il s'agit, ni son fabricant.

De notre point de vue d'être humain, la différence entre un four à micro-ondes et un four conventionnel ne semble pas de la plus haute importance, mais envisagez un instant la question du point de vue du four. Les étapes du processus interne mis en œuvre par un four conventionnel sont complètement différentes de celles d'un four à micro-ondes (sans parler d'un four à convection).

Le pouvoir du principe de l'héritage repose sur le fait qu'une sous-classe n'est pas *obligée* d'hériter à l'identique de toutes les méthodes de la classe de base. Une sous-classe peut hériter de l'essence des méthodes de la classe de base tout en réalisant une implémentation différente de leurs détails.

Surcharger une méthode héritée

Plusieurs fonctions peuvent porter le même nom, à condition qu'elles soient différenciées par le nombre et/ou le type de leurs arguments.

Ce n'est qu'une question de surcharge de fonction



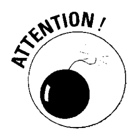
Donner le même nom à deux fonctions (ou plus) s'appelle *surcharger* un nom de fonction.

Les arguments d'une fonction font partie de son nom complet, comme le montre l'exemple suivant :

```
public class MyClass
{
  public static void AFunction()
  {
    // faire quelque chose
  }
  public static void AFunction(int)
  {
    // faire quelque chose d'autre
  }
  public static void AFunction(double d)
  {
    // faire encore quelque chose d'autre
  }
  public static void Main(string[] args)
  {
```

```
    AFunction();
    AFunction(1);
    AFunction(2.0);
}
```

C# peut différencier les méthodes par leurs arguments. Chacun des appels dans `Main()` accède à une fonction différente.



Le type retourné ne fait pas partie du nom complet. Vous pouvez avoir deux fonctions qui ne diffèrent que par le type retourné.

À classe différente, méthode différente

Comme on peut s'y attendre, la classe à laquelle appartient une fonction ou une méthode fait aussi partie de son nom complet. Voyez le segment de code suivant :

```
public class MyClass
{
  public static void AFunction();
  public static void AMethod();
}
public UrClass
{
  public static void AFunction();
  public static void AMethod();
public class Class1
{
  public static void Main(string[] args)
  {
    UrClass.AFunction();
    // invoque la fonction membre MyClass.AMethod()
    MyClass mcObject = new MyClass();
    mcObject.AMethod();
  }
}
```

Le nom de la classe fait partie du nom étendu de la fonction. Il y a le même type de relation entre la fonction `MyClass.AFunction()` et la fonction `UrClass.AFunction()` qu'entre la fonction `MaVoiture. DémarrerMatinHiver()` et la fonction `VotreVoiture.DémarrerMatinHiver()` (avec la vôtre, ça marche).

Redéfinir une méthode d'une classe de base

Ainsi, une méthode d'une classe peut surcharger une autre méthode de la même classe en ayant des arguments différents. De même, une méthode peut aussi surcharger une méthode de sa classe de base. Surcharger une méthode d'une classe de base s'appelle *redéfinir*, ou *cacher* la méthode.

Imaginez que ma banque adopte une politique qui établisse une différence entre les retraits sur les comptes rémunérés et les autres types de retrait. Pour les besoins de notre exemple, imaginez aussi qu'un retrait effectué sur un compte rémunéré coûte une commission de 1,50 F.

Avec l'approche fonctionnelle, vous pourriez implémenter cette politique en définissant dans la classe un indicateur qui dise si l'objet est un `SavingsAccount` ou un simple `BankAccount`. La méthode de retrait devrait alors tester l'indicateur pour savoir si elle doit ou non imputer la commission de 1,50 F :

```
public BankAccount(int nAccountType)
{
  private decimal mBalance;
  private bool isSavingsAccount;
  // indique le solde initial et dit si le compte
  // que vous êtes en train de créer est ou non
  // un compte rémunéré
  public BankAccount(decimal mInitialBalance,
                     bool isSavingsAccount)
  {
    mBalance = mInitialBalance;
    this.isSavingsAccount = isSavingsAccount;
  }
  public decimal Withdraw(decimal mAmount)
  {
    // si le compte est un compte rémunéré . . .
    if (isSavingsAccount)
    {
      // ...alors soustrait 1.50 F
      mBalance -= 1.50M;
    }
    // poursuit avec le même code pour le retrait :
    if (mAmountToWithdraw > mBalance)
    {
      mAmountToWithdraw = mBalance;
    }
    mBalance -= mAmountToWithdraw;
    return mAmountToWithdraw;
```

```
    }
  }
  class MyClass
  {
    public void SomeFunction()
    {
      // Je veux me créer un compte rémunéré :
      BankAccount ba = new BankAccount(0, true);
    }
  }
```

Ma fonction doit dire au constructeur si le compte bancaire est un SavingsAccount en lui passant un indicateur. Le constructeur conserve cet indicateur et l'utilise dans la méthode Withdraw() pour décider s'il faut imputer la commission de 1,50 F.

L'approche orientée objet consiste à redéfinir la méthode Withdraw() dans la classe de base BankAccount, derrière une méthode de même nom, de même taille et de même couleur de cheveux, dans la classe SavingsAccount :

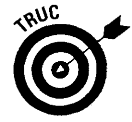



```
// HidingWithdrawal - redéfinit la méthode de retrait de la
//                    classe de base avec une méthode de la
//                    sous-classe du même nom
using System;
namespace HidingWithdrawal
{
  // BankAccount - un compte bancaire très ordinaire
  public class BankAccount
  {
    protected decimal mBalance;
    public BankAccount(decimal mInitialBalance)
    {
      mBalance = mInitialBalance;
    }
    public decimal Balance
    {
      get { return mBalance; }
    }
    public decimal Withdraw(decimal mAmount)
    {
      decimal mAmountToWithdraw = mAmount;
      if (mAmountToWithdraw > mBalance)
      {
        mAmountToWithdraw = mBalance;
      }
      mBalance -= mAmountToWithdraw;
      return mAmountToWithdraw;
```

```
    }
  }
  // SavingsAccount - compte bancaire qui rapporte des intérêts
  public class SavingsAccount : BankAccount
  {
    public decimal mInterestRate;
    // SavingsAccount - lit le taux d'intérêt, exprimé en
    //                  pourcentage (valeur comprise entre 0 et 100)
    public SavingsAccount(decimal mInitialBalance,
                          decimal mInterestRate)
    : base(mInitialBalance)
    {
      this.mInterestRate = mInterestRate / 100;
    }
    // AccumulateInterest - invoquée une fois par période
    public void AccumulateInterest()
    {
      mBalance = mBalance + (mBalance * mInterestRate);
    }
    // Withdraw - tout retrait est autorisé jusqu'à la valeur
    //            du solde ; retourne le montant retiré
    public decimal Withdraw(decimal mWithdrawal)
    {
      // soustrait 1.50 F
      base.Withdraw(1.5M);
      // vous pouvez maintenant effectuer un retrait avec ce qui reste
      return base.Withdraw(mWithdrawal);
    }
  }
  public class Class1
  {
    public static void MakeAWithdrawal(BankAccount ba,
                                       decimal mAmount)
    {
      ba.Withdraw(mAmount);
    }
    public static int Main(string[] args)
    {
      BankAccount ba;
      SavingsAccount sa;
      // crée un compte bancaire, en retire 100 F, et
      // affiche les résultats
      ba = new BankAccount(200M);
      ba.Withdraw(100M);
      // essaie de faire la même chose avec un compte rémunéré
      sa = new SavingsAccount(200M, 12);
      sa.Withdraw(100M);
      // affiche le solde résultant
```

```
            Console.WriteLine("Quand il est invoqué directement :");
            Console.WriteLine("Le solde de BankAccount est {0:C}",
                              ba.Balance);
            Console.WriteLine("Le solde de SavingsAccount est {0:C}",
                              sa.Balance);
            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
            return 0;
        }
    }
}
```

Dans ce cas, la fonction `Main()` crée un objet `BankAccount` avec un solde initial de 200 F, et effectue un retrait de 100 F. `Main()` répète cette opération avec un objet `SavingsAccount`. Lorsque `Main()` effectue le retrait depuis la classe de base, `BankAccount.Withdraw()` effectue cette tâche avec un aplomb remarquable. Lorsque `Main()` retire ensuite 100 F du compte rémunéré, la méthode `SavingsAccount.Withdraw()` fait payer les 1,50 F.



Remarquer que la classe `SavingsAccount.Withdraw` utilise `BankAccount.Withdraw()` plutôt que de manipuler directement le solde. Dans toute la mesure du possible, faites en sorte que ce soit la classe de base qui manipule elle-même ses membres.

En quoi vaut-il mieux redéfinir une méthode qu'ajouter un simple test ?

Vu de l'extérieur, l'ajout d'un indicateur à la méthode `BankAcount.Withdraw()` peut sembler plus simple que le procédé qui consiste à redéfinir une méthode. Après tout, ça ne fait qu'ajouter quatre petites lignes de code, dont deux ne sont que des accolades.

Le problème, c'est en quelque sorte la tuyauterie. Le premier inconvénient est que la classe `BankAccount` n'a aucune raison de se mêler des détails de `SavingsAccount`. Cela violerait notre règle "Rendons à César ce qui est à César", et nous conduit au véritable problème : imaginez que ma banque décide d'ajouter des comptes `CheckingAccount` ou `CDAccount` ou encore `TbillAccount` ? C'est une chose possible, et tous ces types de compte différents seraient associés à des politiques de retrait différentes, chacune nécessitant son propre indicateur. Après l'ajout de trois ou quatre nouveaux types de compte, notre vieille méthode `Withdraw()` commencerait à devenir bien compliquée. Chacune de ces classes devrait plutôt s'occuper elle-même de sa politique de retrait et laisser tranquille notre pauvre vieille `BankAccount.Withdraw()`.

Et si je redéfinis accidentellement une méthode de la classe de base ?

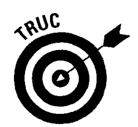
Il peut arriver à tout le monde de redéfinir accidentellement une méthode de la classe de base. Par exemple, je peux avoir une méthode Véhicule.Virage() qui fait tourner le véhicule. Plus tard, quelqu'un étend ma classe Véhicule avec une classe Avion, dont la méthode Virage() est entièrement différente. Il est clair que nous avons là un cas de confusion d'identité. Ces deux méthodes n'ont rien à voir l'une avec l'autre, sinon qu'elles portent le même nom.

Heureusement pour nous, C# sait détecter ce problème.

En compilant l'exemple précédent, HidingWithdraw(), C# génère un avertissement patibulaire. Le texte de ce message est un peu long, mais en voici la partie importante :

```
Le mot-clé new est requis sur
'HidingWithdrawal.SavingsAccount.Withdraw(decimal)', car il
masque le membre hérité
'HidingWithdrawal.BankAccount.Withdraw(decimal)'
```

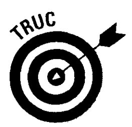
C# essaie de vous dire que vous avez écrit dans une sous-classe une méthode portant le même nom qu'une méthode de la classe de base. Est-ce vraiment ce que vous vouliez faire ?



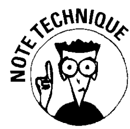
Ce message n'est qu'un avertissement. Vous ne le remarquerez même pas, à moins de passer à la fenêtre Sortie pour voir ce qui y est affiché. Dans presque tous les cas, vous y verrez un avertissement qui vous prévient que quelque chose pourrait bien vous mordre si vous n'y mettez pas bon ordre.

Le descripteur new indique à C# qu'une méthode est redéfinie intentionnellement et que ce n'est pas le résultat d'une négligence :

```
// plus de problèmes avec withdraw()
new public decimal Withdraw(decimal dWithdrawal)
{
  // . . . pas de modifications internes. . .
}
```



Cette utilisation du mot-clé new n'a rien à voir avec l'utilisation du même mot-clé pour créer un objet.



Je me permettrai de faire remarquer ici que c'est l'une des choses que je trouve agaçantes chez C# (et C++ avant lui) : faites ce que vous voulez avec mes méthodes, mais ne surchargez pas mes mots-clés. Quand je dis new, c'est que je veux créer un objet. Ils auraient pu utiliser un autre mot-clé pour indiquer une surcharge intentionnelle.

Revenir à la base

Revenons à la méthode `SavingsAccount.Withdraw()` de l'exemple que nous avons vu plus haut dans ce chapitre. L'appel à `BankAccount.Withdraw()` depuis cette nouvelle méthode contient le mot-clé supplémentaire `base`.

La version suivante de la fonction avec ce mot-clé supplémentaire ne marche pas :

```
new public double Withdraw(double dWithdrawal)
{
  double dAmountWithdrawn = Withdraw(dWithdrawal);
  if (++nNumberOfWithdrawalsThisPeriod > 1)
  {
    dAmountWithdrawn += Withdraw(1.5);
  }
  return dAmountWithdrawn;
}
```

Cet appel a le même problème que celui-ci :

```
void fn()
{
  fn(); // je m'appelle moi-même
}
```

L'appel à `fn()` depuis `fn()` aboutit à s'appeler soi-même, sans fin. De même, un appel de `Withdraw()` à elle-même fait qu'elle s'appelle elle-même en boucle, comme un chat qui court après sa queue, jusqu'à ce que le programme finisse par se planter.

D'une manière ou d'une autre, il vous faut indiquer à C# que l'appel depuis `SavingsAccount.Withdraw()` est là pour invoquer la méthode de la classe de base, `BankAccount.Withdraw()`. Une solution consiste à faire un cast du pointeur `this` dans un objet de la classe `BankAccount` avant d'effectuer l'appel.

```
// Withdraw - cette version accède à la méthode redéfinie dans la classe de
//            base en définissant explicitement le cast de l'objet this
new public double Withdraw(double dWithdrawal)
{
  // cast du pointeur this dans un objet de la classe BankAccount
```

```
    BankAccount ba = (BankAccount)this;
    // invoque Withdraw() en utilisant cet objet BankAccount
    // appelle la fonction BankAccount.Withdraw()
    double dAmountWithdrawn = ba.Withdraw(dWithdrawal);
    if (++nNumberOfWithdrawalsThisPeriod > 1)
    {
      dAmountWithdrawn += ba.Withdraw(1.5);
    }
    return dAmountWithdrawn;
  }
```

Cette solution fonctionne : l'appel `ba.Withdraw()` invoque maintenant la méthode `BankAccount`, comme on le voulait. L'inconvénient de cette approche est la référence explicite à `BankAccount`. Une modification ultérieure du programme pourrait modifier la hiérarchie d'héritage de telle manière que `SavingsAccount` n'hérite plus directement de `BankAccount`. Une telle réorganisation brise cette fonction d'une façon qu'un nouveau programmeur pourra avoir du mal à trouver. Pour moi, je n'arriverais jamais à trouver un bogue comme celui-là.

Il vous faut un moyen de dire à C# d'appeler la fonction `Withdraw()` depuis "la classe qui est juste au-dessus" dans la hiérarchie, sans la nommer explicitement. Ce serait la classe qui est étendue par `SavingsAccount`. C'est dans ce but que C# comporte le mot-clé `base`.



C'est le même mot-clé `base` qu'utilise un constructeur pour passer des arguments au constructeur de la classe de base.

Le mot-clé `base` de C# est la même chose que `this`, mais redéfinit le cast à la classe de base, quelle que soit cette classe :

```
// Withdraw - tout retrait est autorisé jusqu'à la valeur
//            du solde ; retourne le montant retiré
new public decimal Withdraw(decimal mWithdrawal)
{
  // soustrait 1.50 F
  base.Withdraw(1.5M);
  // vous pouvez maintenant effectuer un retrait avec ce qui reste
  return base.Withdraw(mWithdrawal);
}
```

L'appel `base.Withdraw()` invoque maintenant la méthode `BankAccount.Withdraw()`, évitant par-là l'écueil qui consiste à s'invoquer elle-même. En outre, cette solution ne sera pas brisée si la hiérarchie d'héritage est modifiée.

Le polymorphisme

Vous pouvez surcharger une méthode d'une classe de base avec une méthode d'une sous-classe. Aussi simple qu'elle paraisse, cette solution apporte des possibilités considérables, et de ces possibilités viennent des dangers.

Voilà une question difficile : la décision d'appeler BankAccount.Withdraw ou SavingsAccount.Withdraw() doit-elle être prise à la compilation ou à l'exécution ?

Pour faire comprendre la différence, je vais modifier le programme HidingWithdrawal que nous avons vu plus haut d'une manière apparemment inoffensive. Je vais appeler cette nouvelle version HidingWithdrawal-Polymorphically (j'ai allégé le listing en n'y mettant pas ce qui n'a pas changé).




```
public class Class1
{
  public static void MakeAWithdrawal(BankAccount ba,
                                     decimal mAmount)
  {
    ba.Withdraw(mAmount);
  }
  public static int Main(string[] args)
  {
    BankAccount ba;
    SavingsAccount sa;
    ba = new BankAccount(200M);
    MakeAWithdrawal(ba, 100M);
    sa = new SavingsAccount(200M, 12);
    MakeAWithdrawal(sa, 100M);
    // affiche le solde résultant
    Console.WriteLine("\nÉvoqué par un intermédiaire,");
    Console.WriteLine("Le solde de BankAccount est {0:C}", ba.Balance);
    Console.WriteLine("Le solde de SavingsAccount est {0:C}", sa.Balance);
    // attend confirmation de l'utilisateur
    Console.WriteLine("Appuyez sur Entrée pour terminer...");
    Console.Read();
    return 0;
  }
}
```

La sortie de ce programme peut être ou ne pas être déconcertante, selon ce que vous attendiez :

```
Évoqué par un intermédiaire,
Le solde de BankAccount est 100,00 F
Le solde de SavingsAccount est 100,00 F
Appuyez sur Entrée pour terminer...
```

Cette fois, plutôt que d'effectuer un retrait dans `Main()`, le programme passe l'objet compte bancaire à la fonction `MakeAWithdrawal()`.

La première question est dépourvue de mystère : Pourquoi la fonction `MakeAWithdrawal()` accepte-t-elle un objet `SavingsAccount` alors qu'elle dit clairement qu'elle attend un objet `BankAccount` ? La réponse est évidente : "Parce qu'un `SavingsAccount` EST_UN `BankAccount`."

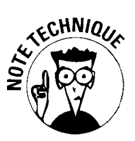
La deuxième question est plus subtile. Quand il lui est passé un objet `BankAccount`, `MakeAWithdrawal()` invoque `BankAccount.Withdraw()`. C'est assez clair. Mais lorsqu'il lui est passé un objet `SavingsAccount`, `MakeAWithdrawal()` appelle la même méthode. Ne devrait-elle pas invoquer la méthode `Withdraw()` dans la sous-classe ?

Le procureur veut montrer que l'appel à `ba.Withdraw()` devrait invoquer la méthode `BankAccount.Withdraw()`. Il est clair que l'objet `ba` est un `BankAccount`. Faire autre chose ne pourrait que faire naître la confusion. Mais la défense a des témoins dans `Main()` pour prouver que bien que l'objet `ba` soit déclaré comme `BankAccount`, c'est en fait un `SavingsAccount`. Le jury ne s'y retrouve plus. Les deux arguments sont tout aussi valides l'un que l'autre.

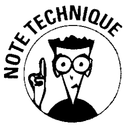
Dans ce cas, C# se range du côté du procureur. Le choix le plus sûr est de s'en tenir au type déclaré, parce qu'il évite toute erreur de communication. L'objet est donc déclaré être un `BankAccount`, et la cause est entendue.

Qu'y a-t-il de mal à utiliser chaque fois le type déclaré ?

Dans certains cas, vous ne voudrez pas utiliser le type déclaré. Ce que vous voudrez vraiment, c'est effectuer l'appel sur la base du *type réel*, c'est-à-dire le type à l'exécution, par opposition au type déclaré. Cette possibilité de décider à l'exécution s'appelle *polymorphisme*, ou *late binding* (liaison tardive). Utiliser le type déclaré s'appelle *early binding* (liaison précoce).

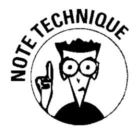


Le terme polymorphisme vient du grec : *poly* signifie plusieurs, *morph* signifie forme, et *isme* signifie peut-être quelque chose en grec.



Polymorphisme et late binding ne sont pas exactement la même chose, mais la différence est subtile. La notion de *polymorphisme* se réfère à la possibilité de décider à l'exécution qu'elle méthode invoquer. La notion de *late binding* se réfère à la manière dont un langage implémente le polymorphisme.

Le polymorphisme est un aspect crucial de la puissance de la programmation orientée objet. Il est si important qu'un langage qui ne le comporte pas ne peut pas être présenté comme un langage orienté objet.



Un langage qui comporte des classes mais pas le polymorphisme est appelé langage à base objets. Le langage Ada en est un exemple.

Sans le polymorphisme, l'héritage n'aurait guère de signification. Je vais donner encore un exemple pour vous le montrer. Imaginez que j'aie écrit ce programme à succès mondial qui utilisait une classe nommé `Student`. Après quelques mois de conception, de codage et de test, je publie cette application pour récolter les avis de mes collègues et des critiques en tout genre (on a même parlé de créer une nouvelle catégorie de prix Nobel pour le logiciel, mais par modestie jai ignoré ces suggestions).

Le temps passe, et mon patron me demande d'ajouter à ce programme la prise en compte des étudiants diplômés, qui ne sont pas tout à fait identiques aux étudiants ordinaires (ils revendiquent sans doute de ne pas être identiques du tout). Supposez que la formule de calcul des frais de scolarité soit complètement différente de celle utilisée pour un étudiant ordinaire. Mais mon patron ne sait pas et ne veut pas savoir qu'à ce niveau de profondeur du programme il y a de nombreux appels à la fonction `calcTuition()`.

```
void SomeFunction(Student s)
{
  // . . . quoi qu'elle puisse faire. . .
  s.CalcTuition();
  // . . . continue . . .
}
```

Si C# n'admettait pas le late binding, il me faudrait passer en revue le code de `someFunction()` pour vérifier si l'objet `student` qui lui est passé est un `GraduateStudent` ou un `Student`. Le programme appellerait `Student.CalcTuition()` lorsque `s` est un `Student`, et `GraduateStudent.CalcTuition()` lorsque c'est un étudiant diplômé.

Tout cela ne se présente pas mal, sauf pour trois choses. Pour commencer, ce n'est là qu'une fonction. Supposez que `calcTuition()` soit appelée depuis de nombreux endroits. Supposez aussi que `calcTuition()` ne soit pas la seule différence entre les deux classes. Mes chances de trouver tous les endroits qui doivent être modifiés ne sont pas des plus élevées.

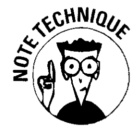
Avec le polymorphisme, je peux laisser C# décider de la méthode à appeler.

Accéder par le polymorphisme à une méthode redéfinie en utilisant `is`

Comment rendre mon programme polymorphe ? C# offre une approche pour résoudre le problème manuellement avec un tout nouveau mot-clé : `is`. L'expression `ba is SavingsAccount` retourne `true` ou `false` selon la classe de l'objet à l'exécution. Le type déclaré pourrait être `BankAccount`, mais quel est-il en réalité ?

```
public class Class1
{
  public static void MakeAWithdrawal(BankAccount ba,
                                     decimal mAmount)
  {
    if ba is SavingsAccount
    {
      SavingsAccount sa = (SavingsAccount)ba;
      sa.Withdraw(mAmount);
    } else
    {
      ba.Withdraw(mAmount);
    }
  }
}
```

Maintenant, quand `Main()` passe à la fonction un objet `SavingsAccount`, `MakeAWithdrawal()` vérifie à l'exécution le type de l'objet `ba` et invoque `SavingsAccount.Withdraw()`.



Au passage, je vous signale que le programmeur aurait pu réaliser le cast et l'appel dans une même ligne : `((SavingsAccount)ba).Withdraw(mAmount)`. Je ne mentionne la chose que parce que vous la verrez beaucoup dans des programmes écrits par des gens qui aiment faire de l'esbroufe.

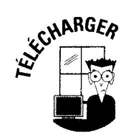
En fait, l'approche "is" fonctionne, mais c'est vraiment une mauvaise idée. Elle nécessite que SomeFunction() connaisse tous les différents types d'étudiants et quels sont ceux qui sont représentés par les différentes classes. Cela fait peser une trop lourde responsabilité sur les épaules de la pauvre vieille SomeFunction(). Pour le moment, mon application ne connaît que deux types de comptes bancaires, mais supposez que mon patron me demande d'en implémenter un nouveau, CheckingAccount, et que celui-ci soit associé à une autre politique de Withdraw(). Mon programme ne fonctionnera pas correctement si je ne trouve pas toutes les fonctions qui testent à l'exécution le type de cet argument.

Déclarer une méthode comme virtuelle

En tant qu'auteur de SomeFunction(), je ne veux pas connaître tous les différents types de compte. Je veux que ce soit aux programmeurs qui utilisent SomeFunction() de connaître leurs types de compte, et qu'ils me laissent tranquille avec ça. Je veux que ce soit C# qui prenne les décisions sur les méthodes à invoquer en fonction du type de l'objet à l'exécution.

Pour dire à C# de faire le choix à l'exécution de la version de Withdrawal() à utiliser, je marque la fonction de la classe de base avec le mot-clé virtual, et la fonction de chaque sous-classe avec le mot-clé override.

J'ai réécrit l'exemple de programme précédent en utilisant le polymorphisme. J'ai ajouté des instructions de sortie aux méthodes Withdraw() pour montrer que ce sont effectivement les bonnes méthodes qui sont invoquées (j'ai supprimé ce qui faisait double emploi pour ne pas vous ennuyer avec des choses inutiles). Voici donc le programme PolymorphicInheritance :



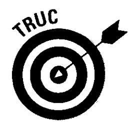

```
// PolymorphicInheritance - utilise le polymorphisme pour
//                        redéfinir une méthode dans la classe de base
using System;
namespace PolymorphicInheritance
{
  // BankAccount - un compte bancaire très ordinaire
  public class BankAccount
  {
    // . . . la même chose ici . . .
    public virtual decimal Withdraw(decimal mAmount)
    {
      decimal mAmountToWithdraw = mAmount;
      if (mAmountToWithdraw > mBalance)
      {
```

```
            mAmountToWithdraw = mBalance;
          }
          mBalance -= mAmountToWithdraw;
          return mAmountToWithdraw;
        }
      }
      // SavingsAccount - compte bancaire qui rapporte des intérêts
      public class SavingsAccount : BankAccount
      {
        // . . . la même chose ici aussi . . .
        // Withdraw - tout retrait est autorisé jusqu'à la valeur
        //            du solde ; retourne le montant retiré
        override public decimal Withdraw(decimal mWithdrawal)
        {
          // soustrait 1.50 F
          base.Withdraw(1.5M);
          // vous pouvez maintenant effectuer un retrait avec ce qui reste
          return base.Withdraw(mWithdrawal);
        }
      }
      public class Class1
      {
        public static void MakeAWithdrawal(BankAccount ba,
                                           decimal mAmount)
        {
          ba.Withdraw(mAmount);
        }
        public static void Main(string[] args)
        {
          // . .. pas de changement ici non plus . . .
        }
      }
    }
```

L'exécution de ce programme donne la sortie suivante :

```
Évoqué par un intermédiaire,
Le solde de BankAccount est 100,00 F
Le solde de SavingsAccount est 98,50 F
Appuyez sur Entrée pour terminer...
```

La fonction `Withdraw()` est marquée comme `virtual` dans la classe de base `BankAccount`, alors que la méthode `Withdraw()` de la sous-classe est marquée avec le mot-clé `override`. Bien que la méthode `MakeAWithdrawal()` soit inchangée, le programme donne une sortie différente parce que l'appel `ba.Withdraw()` est résolu sur la base du type de `ba` à l'exécution.



Pour vous faire une idée précise de la manière dont tout cela fonctionne, il est nécessaire que vous exécutiez le programme dans le débogueur de Visual Studio. Générez le programme comme d'habitude, et appuyez sur la touche F11 autant de fois que nécessaire pour faire avancer le programme pas à pas. Il est impressionnant de voir le même appel aboutir à une méthode ou à une autre selon le moment où il se produit.

La période abstraite de C#

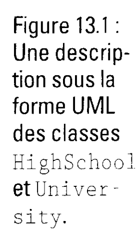
À ce que je sais, un canard est un type d'oiseau, de même qu'un colibri et une hirondelle. En fait, tout oiseau est un sous-type d'oiseau. La réciproque de ce principe est qu'il n'existe pas d'oiseau qui ne soit pas un sous-type d'oiseau. Cette remarque ne paraît pas très profonde, mais en fait, elle l'est. L'équivalent logiciel de cet énoncé est que tout objet `oiseau` est une instance d'une certaine sous-classe de `Oiseau`. Il n'y a pas d'instance de la classe `Oiseau`.

Il y a divers types d'oiseaux qui partagent de nombreuses propriétés (sinon, ils ne seraient pas des oiseaux), mais il n'y a pas deux types dont toutes les propriétés sont les mêmes (sinon, ce ne serait pas deux types différents). Pour prendre un exemple particulièrement simple, tous les oiseaux n'ont pas la même manière de `Voler()`. Le canard a son style, l'hirondelle aussi. Ils ont beaucoup de points communs, mais ce n'est pas exactement la même chose. Le style du colibri est complètement différent. Ne me posez pas de questions sur les émeus et les autruches.

Mais si tous les oiseaux n'ont pas la même manière de voler, alors, qu'est-ce que `Oiseau.Voler()` ? La réponse est simple : c'est le sujet de cette section.

Le factoring entre classes

Les gens produisent des taxonomies d'objets en les regroupant sur la base des propriétés qu'ils partagent. Pour comprendre comment fonctionne la classification, considérez les classes `HighSchool` et `University`, comme le montre la Figure 13.1. Cette figure utilise UML (Unified Modeling Language), qui est un langage graphique, pour décrire une classe en même temps que les relations de celle-ci avec d'autres.



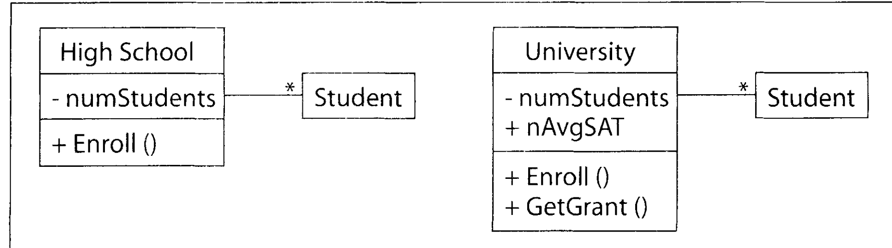
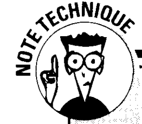


Figure 13.1 : Une description sous la forme UML des classes `HighSchool` et `University`.



Le langage UML

Le langage UML (Unified Modeling Language) est un langage de modélisation, permettant de définir clairement une grande partie des relations entre des objets dans un programme. L'un des avantages d'UML est que l'on peut en ignorer les aspects les plus spécialisés sans en perdre entièrement la signification.

Les caractéristiques de base d'UML sont les suivantes :

- Une classe est représentée par un rectangle, divisé verticalement en trois sections. Le nom de la classe apparaît dans la première section en partant du haut.
- Les membres donnée de la classe apparaissent dans la section du milieu, et les méthodes dans la section du bas. Si la classe n'a pas de membres donnée ou de méthodes, vous pouvez omettre la section du milieu ou celle du bas.
- Les membres précédés par un signe + sont publics, et ceux qui sont précédés par un signe - sont privés. UML ne dispose pas de symboles pour représenter la visibilité et la protection.

  Un membre privé n'est accessible qu'à d'autres membres de la même classe. Un membre public est accessible à toutes les classes.
- Le symbole "{abstract}" à côté d'un nom indique que la classe ou la méthode est abstraite.

  UML utilise en fait un symbole différent pour une méthode abstraite, mais je simplifie.

- Une flèche entre deux classes représente une relation entre ces deux classes. Un nombre au-dessus du trait exprime la cardinalité. Le symbole "*" signifie *un nombre quelconque*. Si aucun nombre n'est présent, la cardinalité est supposée égale à 1. Ainsi, dans la Figure 13.1, vous pouvez voir qu'une université peut avoir un nombre quelconque d'étudiants.

- Un trait se terminant par une grande flèche largement ouverte exprime la relation EST_UN (l'héritage). D'autres types de relations sont également représentés, dont la relation A_UN.



Un objet Voiture EST_UN Véhicule, mais un objet Voiture A_UN Moteur.

Vous pouvez voir dans la Figure 13.1 que les high schools et les universités ont en commun plusieurs propriétés similaires (en fait, bien plus qu'on ne pourrait le penser). Ces deux types d'établissement offrent une méthode `Enroll()`, publiquement disponible, pour ajouter de nouveaux objets `Student`. En outre, l'un et l'autre comportent un objet privé `numStudents` qui contient le nombre d'étudiants de l'établissement. Enfin, l'une des caractéristiques communes est la relation entre les étudiants : un établissement peut avoir un nombre quelconque d'étudiants, mais un étudiant ne peut faire partie que d'un seul établissement à la fois. La plupart des universités, et même certaines high schools, offrent plus que ce que je viens de décrire, mais il me suffit d'un type de chaque.

En plus des caractéristiques d'une high school, l'université contient une méthode `GetGrant()` et un membre donnée `nAvgSAT`. L'entrée dans une high school ne nécessite pas l'examen SAT (Scholastic Aptitude Test), et on ne peut y obtenir de prêt fédéral (à moins que je ne sois allé dans les mauvaises high schools).

La Figure 13.1 est tout à fait correcte, mais beaucoup d'informations s'y trouvent dupliquées. On pourrait réduire cette duplication en permettant à la classe la plus compliquée, `University`, d'hériter de la classe plus simple `HighSchool`, comme le montre la Figure 13.2.

La classe `HighSchool` est inchangée, mais la classe `University` est plus facile à décrire. Nous disons que "une `University` est une `HighSchool` qui possède aussi un objet `nAvgSAT` et une méthode `GetGrant()`", mais cette solution comporte un problème fondamental : une université n'est pas une high school qui... quoi que ce soit.

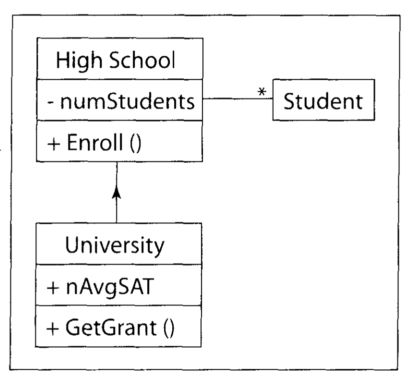


Figure 13.2 : L'héritage de `HighSchool` simplifie la classe `University`, mais il introduit quelques problèmes.

Vous me direz : "Et alors ? l'héritage fonctionne, et ça fait du travail en moins." C'est vrai, mais les réserves que je fais ne sont pas que de triviales questions de style. De fausses représentations de ce genre sont sources de confusion pour le programmeur, dans l'immédiat comme par la suite. Un jour, un programmeur qui ne connaîtra pas mes astuces de programmation devra lire mon code et comprendre ce qu'il fait. Une représentation fausse est difficile à comprendre et à utiliser.

En outre, de telles représentations fausses peuvent conduire ultérieurement à des problèmes. Imaginez que la high school décide de nommer un étudiant "favori" pour son banquet annuel (usage local). Le programmeur, astucieux, ajoute alors à la classe `HighSchool` la méthode `NameFavorite()`, que l'application invoque pour nommer favori l'objet `Student` correspondant.

Mais maintenant, j'ai un problème. La plupart des universités n'ont pas pour pratique de nommer quoi que ce soit favori, mais aussi longtemps que `University` hérite de `HighSchool`, elle hérite de la méthode `NameFavorite()`. Une méthode supplémentaire peut sembler sans importance. Vous pourriez dire : "Il suffit de l'ignorer."

Une méthode de plus n'a pas grande importance, mais c'est une pierre de plus dans le mur de la confusion. Avec le temps, les méthodes et les propriétés supplémentaires s'accumulent, jusqu'à ce que la classe `University` se trouve bien encombrée de tous ces bagages. Ayez pitié du pauvre développeur de logiciel qui doit comprendre quelles méthodes sont "véritables" et lesquelles ne le sont pas.

Un tel "héritage de complaisance" conduit à un autre problème. À la manière dont elle est écrite, la Figure 13.2 implique qu'une `University` et une `HighSchool` ont la même procédure de recrutement. Si peu vraisemblable que cela paraisse, supposez que ce soit vrai. Le programme est développé, emballé

et expédié au public qui n'attend que lui (bien sûr, je n'ai pas oublié d'y mettre le nombre de bogues nécessaires pour que tout le monde veuille s'offrir, moyennant un prix tout à fait raisonnable, la mise à jour vers la version 2).

Quelques mois passent, et l'établissement décide de modifier sa procédure de recrutement. Il ne sera pas évident pour tout le monde qu'en modifiant la procédure de recrutement de la high school c'est aussi la procédure d'inscription au collège voisin qui a été modifiée.

Comment faire pour éviter un tel problème ? Une solution est de ne pas aller à l'école, mais une autre consiste à corriger la source du problème : une université n'est pas un type particulier de high school. Il existe bien une relation entre les deux, mais cette relation n'est pas EST_UN. Au contraire, universités et high schools sont deux types différents d'établissement scolaire.

La Figure 13.3 décrit cette relation. La classe `School` nouvellement définie contient les propriétés communes des deux types d'établissement, y compris leurs relations avec les objets `Student`. `School` contient même la méthode commune `Enroll()`, bien qu'elle soit abstraite car `HighSchool` et `University` ne l'implémentent pas de la même manière.

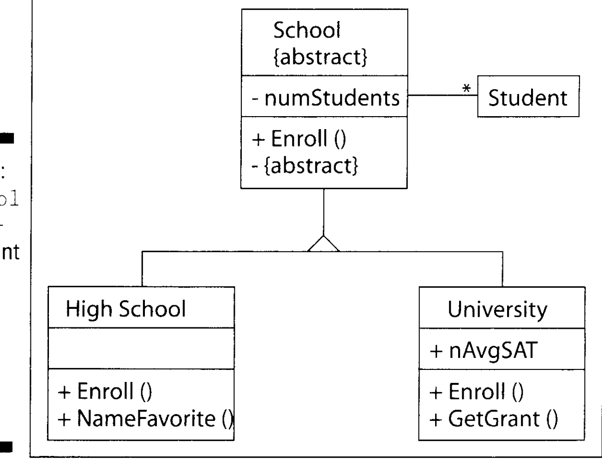
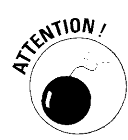


Figure 13.3 : `HighSchool` et `University` doivent l'une et l'autre être basées sur une classe commune `School`.

Les classes `HighSchool` et `University` héritent maintenant toutes deux d'une classe de base commune. Chacune contient des membres qui lui sont propres : `NameFavorite()` dans le cas de `HighSchool`, et `GetGrant()` pour `University`. De plus, ces deux classes substituent à la méthode `Enroll()` une redéfinition de celle-ci décrivant le mode de recrutement de chaque type d'établissement.

L'introduction de la classe `School` présente au moins deux gros avantages. Le premier est de correspondre à la réalité. Une `University` est une `School`, mais ce n'est pas une `HighSchool`. Correspondre à la réalité, c'est bien, mais ce n'est pas suffisant. Le deuxième avantage est d'isoler chaque classe des modifications apportées à l'autre. Quand mon patron viendra me voir un peu plus tard, ce qui se produira sans aucun doute, pour me demander d'introduire le discours de bienvenue à l'université, je pourrai ajouter la méthode `CommencementSpeech()` à la classe `University`, sans affecter la classe `HighSchool`.

Ce processus qui consiste à externaliser les propriétés communes de classes similaires s'appelle *factoring*. C'est une caractéristique importante des langages orientés objet, pour les raisons que nous avons décrites plus haut, plus une nouvelle : la réduction de la redondance. Je vais me répéter : la redondance ne peut faire que du mal. Ne la laissez jamais entrer.



Le factoring n'est légitime que si la relation d'héritage correspond à la réalité. Son application à une classe `Souris` et une classe `Joystick` parce que l'un comme l'autre est un dispositif matériel de pointage est légitime. Son application à une classe `Souris` et une classe `Affichage` parce que l'un comme l'autre fait des appels de bas niveau au système d'exploitation ne l'est pas.

Le factoring peut produire plusieurs niveaux d'abstraction, et en général, c'est le cas. Par exemple, un programme écrit pour une hiérarchie plus complète d'établissements scolaires pourrait avoir une structure de classes plus proche de celle montrée par la Figure 13.4.

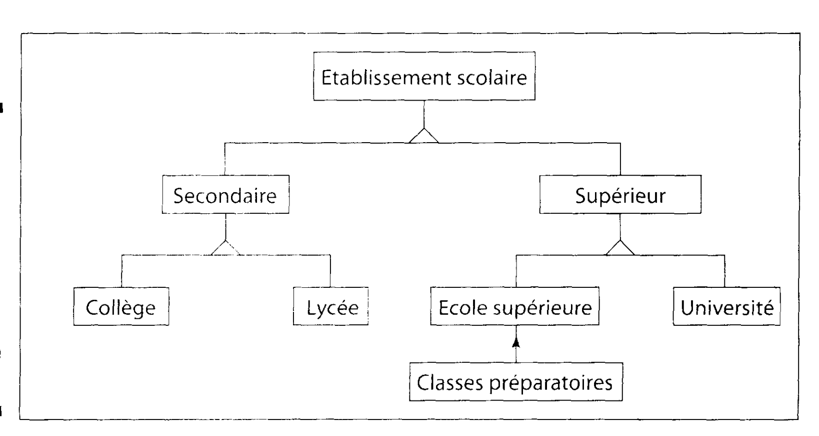


Figure 13.4 : Le factoring produit généralement des couches supplémentaires dans la hiérarchie d'héritage.

Vous pouvez voir que j'ai inséré une nouvelle classe entre `University` et `School` : `HigherLearning`. J'ai subdivisé cette nouvelle classe en `College` et

University. Ce type de hiérarchie de classes à plusieurs niveaux est courant et souhaitable lorsque l'on met des relations en facteurs communs. Il correspond à la réalité et il pourra parfois vous suggérer des solutions subtiles à un problème.

Remarquez toutefois qu'il n'y a pas de Théorie unifiée du factoring applicable à n'importe quel ensemble de classes. Les relations montrées par la Figure 13.4 semblent naturelles, mais supposez qu'une application différencie plutôt les types d'établissements scolaires selon qu'ils sont administrés ou non par des élus locaux. Les relations correspondantes, montrées par la Figure 13.5, conviennent mieux à ce type de problème.

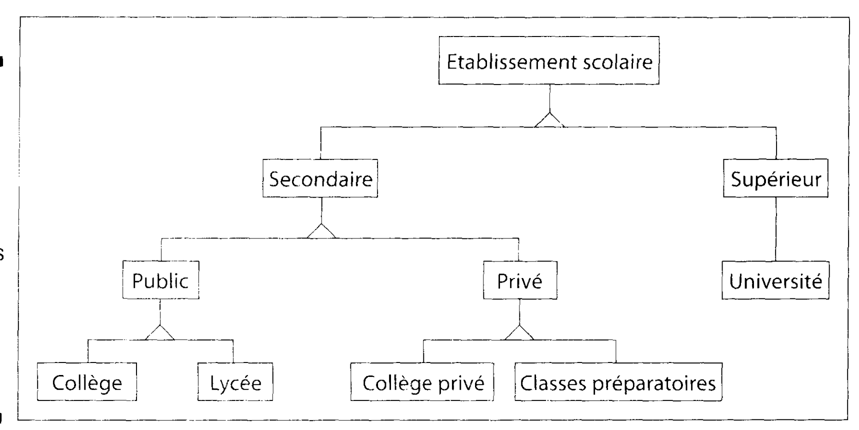


Figure 13.5 : Il n'y a pas de factoring "universel". La manière appropriée de définir les classes dépend en partie du problème à résoudre.

Il ne me reste qu'un concept : la classe abstraite

Si intellectuellement satisfaisant que puisse être le factoring, il introduit un problème qui lui est propre. Revenez encore une fois à BankAccount. Pensez un instant à la manière dont vous pourriez définir les différentes fonctions membre qui y sont définies.

La plupart des fonctions membre de BankAccount ne sont pas un problème, car elles sont implémentées de la même manière par les deux types de compte. C'est avec BankAccount que vous devez implémenter ces fonctions communes. Withdraw(), quant à elle, est différente. Les règles de retrait d'un compte rémunéré sont différentes de celles d'un compte chèque. Vous aurez donc à implémenter SavingsAccount.Withdrawal() différemment de CheckingAccount.Withdraw(). Mais comment implémenter BankAccount.Withdrawal() ?

Si vous demandez son aide au directeur de la banque, j'imagine que la conversation pourrait ressembler à ceci :

"Quelles sont les règles pour effectuer un retrait sur un compte ?" demandez-vous, plein d'espoir.

"Quel type de compte ? Un compte chèque ou un compte rémunéré ?"

"Un compte, répondez-vous, simplement un compte."

Regard vague et désespéré.

Le problème est que cette question n'a pas de sens. Il n'y a pas de compte qui soit "simplement un compte". Tous les comptes (dans cet exemple) sont soit des comptes chèque, soit des comptes rémunérés. La notion de compte est une notion abstraite qui rassemble les propriétés communes aux deux classes correspondant à une réalité concrète. Elle est incomplète, car il lui manque la propriété critique `Withdraw()` (en allant plus loin dans les détails, vous trouverez peut-être d'autres propriétés qui font défaut à un simple compte).

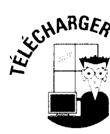
Le concept de `BankAccount` est un concept abstrait.

Comment utiliser une classe abstraite ?

Une classe abstraite sert à décrire des concepts abstraits.

Une *classe abstraite* est une classe comportant une ou plusieurs méthodes abstraites. Une méthode abstraite est une méthode déclarée `abstract`. Allons plus loin : une méthode abstraite n'a pas d'implémentation. Vous êtes maintenant dans le brouillard.

Considérez ce programme de démonstration, allégé pour la circonstance :




```
// AbstractInheritance - la classe BankAccount est vraiment
//                       abstraite parce qu'il n'existe pas
//                       d'implémentation unique pour Withdraw
namespace AbstractInheritance
{
  using System;
  // AbstractBaseClass - crée une classe abstraite de contenant rien d'autre
  //                     qu'une méthode Output()
  abstract public class AbstractBaseClass
  {
```

Si vous demandez son aide au directeur de la banque, j'imagine que la conversation pourrait ressembler à ceci :

"Quelles sont les règles pour effectuer un retrait sur un compte ?" demandez-vous, plein d'espoir.

"Quel type de compte ? Un compte chèque ou un compte rémunéré ?"

"Un compte, répondez-vous, simplement un compte."

Regard vague et désespéré.

Le problème est que cette question n'a pas de sens. Il n'y a pas de compte qui soit "simplement un compte". Tous les comptes (dans cet exemple) sont soit des comptes chèque, soit des comptes rémunérés. La notion de compte est une notion abstraite qui rassemble les propriétés communes aux deux classes correspondant à une réalité concrète. Elle est incomplète, car il lui manque la propriété critique `Withdraw()` (en allant plus loin dans les détails, vous trouverez peut-être d'autres propriétés qui font défaut à un simple compte).

Le concept de `BankAccount` est un concept abstrait.

Comment utiliser une classe abstraite ?

Une classe abstraite sert à décrire des concepts abstraits.

Une *classe abstraite* est une classe comportant une ou plusieurs méthodes abstraites. Une méthode abstraite est une méthode déclarée `abstract`. Allons plus loin : une méthode abstraite n'a pas d'implémentation. Vous êtes maintenant dans le brouillard.

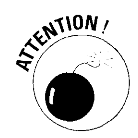
Considérez ce programme de démonstration, allégé pour la circonstance :




```
// AbstractInheritance - la classe BankAccount est vraiment
//                       abstraite parce qu'il n'existe pas
//                       d'implémentation unique pour Withdraw
namespace AbstractInheritance
{
  using System;
  // AbstractBaseClass - crée une classe abstraite de contenant rien d'autre
  //                     qu'une méthode Output()
  abstract public class AbstractBaseClass
  {
```

L'introduction de la classe `School` présente au moins deux gros avantages. Le premier est de correspondre à la réalité. Une `University` est une `School`, mais ce n'est pas une `HighSchool`. Correspondre à la réalité, c'est bien, mais ce n'est pas suffisant. Le deuxième avantage est d'isoler chaque classe des modifications apportées à l'autre. Quand mon patron viendra me voir un peu plus tard, ce qui se produira sans aucun doute, pour me demander d'introduire le discours de bienvenue à l'université, je pourrai ajouter la méthode `CommencementSpeech()` à la classe `University`, sans affecter la classe `HighSchool`.

Ce processus qui consiste à externaliser les propriétés communes de classes similaires s'appelle *factoring*. C'est une caractéristique importante des langages orientés objet, pour les raisons que nous avons décrites plus haut, plus une nouvelle : la réduction de la redondance. Je vais me répéter : la redondance ne peut faire que du mal. Ne la laissez jamais entrer.



Le factoring n'est légitime que si la relation d'héritage correspond à la réalité. Son application à une classe `Souris` et une classe `Joystick` parce que l'un comme l'autre est un dispositif matériel de pointage est légitime. Son application à une classe `Souris` et une classe `Affichage` parce que l'un comme l'autre fait des appels de bas niveau au système d'exploitation ne l'est pas.

Le factoring peut produire plusieurs niveaux d'abstraction, et en général, c'est le cas. Par exemple, un programme écrit pour une hiérarchie plus complète d'établissements scolaires pourrait avoir une structure de classes plus proche de celle montrée par la Figure 13.4.

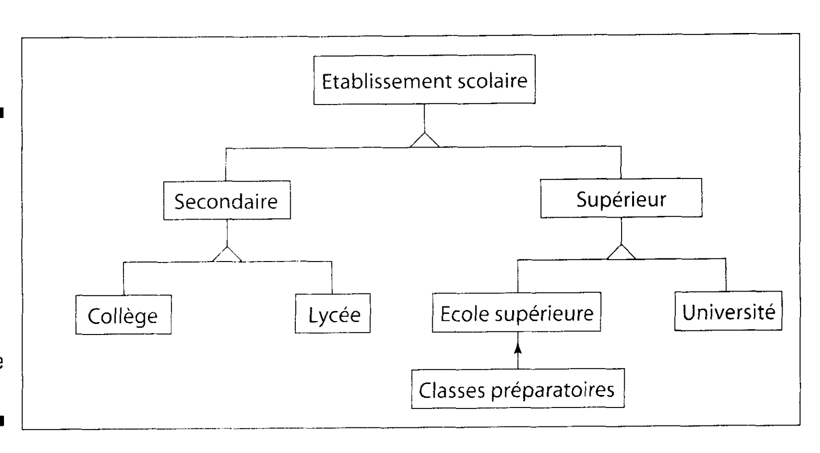


Figure 13.4 : Le factoring produit généralement des couches supplémentaires dans la hiérarchie d'héritage.

Vous pouvez voir que j'ai inséré une nouvelle classe entre `University` et `School` : `HigherLearning`. J'ai subdivisé cette nouvelle classe en `College` et

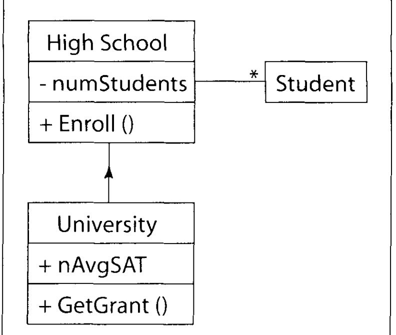



Figure 13.2 : L'héritage de `HighSchool` simplifie la classe `University`, mais il introduit quelques problèmes.

Vous me direz : "Et alors ? l'héritage fonctionne, et ça fait du travail en moins." C'est vrai, mais les réserves que je fais ne sont pas que de triviales questions de style. De fausses représentations de ce genre sont sources de confusion pour le programmeur, dans l'immédiat comme par la suite. Un jour, un programmeur qui ne connaîtra pas mes astuces de programmation devra lire mon code et comprendre ce qu'il fait. Une représentation fausse est difficile à comprendre et à utiliser.

En outre, de telles représentations fausses peuvent conduire ultérieurement à des problèmes. Imaginez que la high school décide de nommer un étudiant "favori" pour son banquet annuel (usage local). Le programmeur, astucieux, ajoute alors à la classe `HighSchool` la méthode `NameFavorite()`, que l'application invoque pour nommer favori l'objet `Student` correspondant.

Mais maintenant, j'ai un problème. La plupart des universités n'ont pas pour pratique de nommer quoi que ce soit favori, mais aussi longtemps que `University` hérite de `HighSchool`, elle hérite de la méthode `NameFavorite()`. Une méthode supplémentaire peut sembler sans importance. Vous pourriez dire : "Il suffit de l'ignorer."

Une méthode de plus n'a pas grande importance, mais c'est une pierre de plus dans le mur de la confusion. Avec le temps, les méthodes et les propriétés supplémentaires s'accumulent, jusqu'à ce que la classe `University` se trouve bien encombrée de tous ces bagages. Ayez pitié du pauvre développeur de logiciel qui doit comprendre quelles méthodes sont "véritables" et lesquelles ne le sont pas.

Un tel "héritage de complaisance" conduit à un autre problème. À la manière dont elle est écrite, la Figure 13.2 implique qu'une `University` et une `HighSchool` ont la même procédure de recrutement. Si peu vraisemblable que cela paraisse, supposez que ce soit vrai. Le programme est développé, emballé



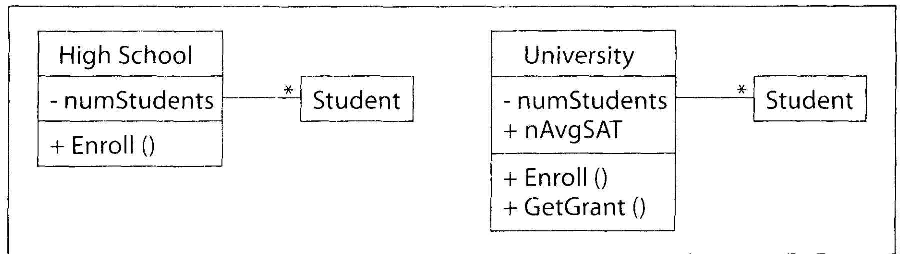
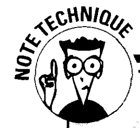


Figure 13.1 : Une description sous la forme UML des classes `HighSchool` et `University`.



Le langage UML

Le langage UML (Unified Modeling Language) est un langage de modélisation, permettant de définir clairement une grande partie des relations entre des objets dans un programme. L'un des avantages d'UML est que l'on peut en ignorer les aspects les plus spécialisés sans en perdre entièrement la signification.

Les caractéristiques de base d'UML sont les suivantes :

- Une classe est représentée par un rectangle, divisé verticalement en trois sections. Le nom de la classe apparaît dans la première section en partant du haut.
- Les membres donnée de la classe apparaissent dans la section du milieu, et les méthodes dans la section du bas. Si la classe n'a pas de membres donnée ou de méthodes, vous pouvez omettre la section du milieu ou celle du bas.
- Les membres précédés par un signe + sont publics, et ceux qui sont précédés par un signe - sont privés. UML ne dispose pas de symboles pour représenter la visibilité et la protection.

  Un membre privé n'est accessible qu'à d'autres membres de la même classe. Un membre public est accessible à toutes les classes.
- Le symbole "{abstract}" à côté d'un nom indique que la classe ou la méthode est abstraite.

  UML utilise en fait un symbole différent pour une méthode abstraite, mais je simplifie.

```
        mAmountToWithdraw = mBalance;
      }
      mBalance -= mAmountToWithdraw;
      return mAmountToWithdraw;
    }
  }
  // SavingsAccount - compte bancaire qui rapporte des intérêts
  public class SavingsAccount : BankAccount
  {
    // . . . la même chose ici aussi . . .
    // Withdraw - tout retrait est autorisé jusqu'à la valeur
    //            du solde ; retourne le montant retiré
    override public decimal Withdraw(decimal mWithdrawal)
    {
      // soustrait 1.50 F
      base.Withdraw(1.5M);
      // vous pouvez maintenant effectuer un retrait avec ce qui reste
      return base.Withdraw(mWithdrawal);
    }
  }
  public class Class1
  {
    public static void MakeAWithdrawal(BankAccount ba,
                                       decimal mAmount)
    {
      ba.Withdraw(mAmount);
    }
    public static void Main(string[] args)
    {
      // . .. pas de changement ici non plus . . .
    }
  }
}
```

L'exécution de ce programme donne la sortie suivante :

```
Évoqué par un intermédiaire,
Le solde de BankAccount est 100,00 F
Le solde de SavingsAccount est 98,50 F
Appuyez sur Entrée pour terminer...
```

La fonction `Withdraw()` est marquée comme `virtual` dans la classe de base `BankAccount`, alors que la méthode `Withdraw()` de la sous-classe est marquée avec le mot-clé `override`. Bien que la méthode `MakeAWithdrawal()` soit inchangée, le programme donne une sortie différente parce que l'appel `ba.Withdraw()` est résolu sur la base du type de `ba` à l'exécution.

Tout cela ne se présente pas mal, sauf pour trois choses. Pour commencer, ce n'est là qu'une fonction. Supposez que `calcTuition()` soit appelée depuis de nombreux endroits. Supposez aussi que `calcTuition()` ne soit pas la seule différence entre les deux classes. Mes chances de trouver tous les endroits qui doivent être modifiés ne sont pas des plus élevées.

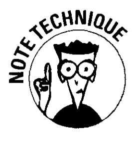
Avec le polymorphisme, je peux laisser C# décider de la méthode à appeler.

Accéder par le polymorphisme à une méthode redéfinie en utilisant `is`

Comment rendre mon programme polymorphe ? C# offre une approche pour résoudre le problème manuellement avec un tout nouveau mot-clé : `is`. L'expression `ba is SavingsAccount` retourne `true` ou `false` selon la classe de l'objet à l'exécution. Le type déclaré pourrait être `BankAccount`, mais quel est-il en réalité ?

```
public class Class1
{
  public static void MakeAWithdrawal(BankAccount ba,
                                     decimal mAmount)
  {
    if ba is SavingsAccount
    {
      SavingsAccount sa = (SavingsAccount)ba;
      sa.Withdraw(mAmount);
    } else
    {
      ba.Withdraw(mAmount);
    }
  }
}
```

Maintenant, quand `Main()` passe à la fonction un objet `SavingsAccount`, `MakeAWithdrawal()` vérifie à l'exécution le type de l'objet `ba` et invoque `SavingsAccount.Withdraw()`.



Au passage, je vous signale que le programmeur aurait pu réaliser le cast et l'appel dans une même ligne : `((SavingsAccount)ba).Withdraw(mAmount)`. Je ne mentionne la chose que parce que vous la verrez beaucoup dans des programmes écrits par des gens qui aiment faire de l'esbroufe.

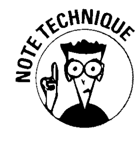
EST_UN Int32 en utilisant le mot-clé is. La sortie de cette portion du programme se présente ainsi :

```
Extrait d'une liste les nombres entiers
L'élément numéro 1 est 2
L'élément numéro 3 est 4
```

Le programme termine son numéro de cirque en utilisant une fois de plus la descendance d'Object. Toutes les sous-classes d'Object (c'est-à-dire toutes les classes) implémentent ToString(). Par conséquent, si nous voulons simplement afficher les membres du tableau d'objets, nous n'avons absolument pas besoin de nous préoccuper de leur type. La section finale de Main() effectue encore une boucle sur les objets du tableau, demandant cette fois à chaque objet de se mettre en forme lui-même en utilisant sa méthode ToString(). Les résultats apparaissent ainsi :

```
Affichage de tous les objets de la liste
Objets[0] est <this is a string>
Objets[1] est <2>
Objets[2] est <Class1 du programme StructureExample>
Objets[3] est <4>
Objets[4] est <5.5>
Appuyez sur Entrée pour terminer...
```

Comme les animaux sortant de l'Arche de Noé, chaque objet se présente comme le seul de sa catégorie. J'ai implémenté une méthode ToString() triviale pour Class1, rien que pour montrer qu'elle sait jouer avec toutes les autres classes.



En fait, c'est indubitablement ToString() qui permet à Console.Write() d'exécuter son tour de magie. Je ne suis pas allé voir dans le code source, mais je parierais volontiers que Write() accepte ses arguments en tant qu'objets. Elle peut alors invoquer simplement ToString() sur l'objet pour le convertir en un format affichable (en dehors du premier argument, qui peut contenir des indications {n} de contrôle de format).

La sortie de ce programme peut être ou ne pas être déconcertante, selon ce que vous attendiez :

```
Évoqué par un intermédiaire,
Le solde de BankAccount est 100,00 F
Le solde de SavingsAccount est 100,00 F
Appuyez sur Entrée pour terminer...
```

Cette fois, plutôt que d'effectuer un retrait dans `Main()`, le programme passe l'objet compte bancaire à la fonction `MakeAWithdrawal()`.

La première question est dépourvue de mystère : Pourquoi la fonction `MakeAWithdrawal()` accepte-t-elle un objet `SavingsAccount` alors qu'elle dit clairement qu'elle attend un objet `BankAccount` ? La réponse est évidente : "Parce qu'un `SavingsAccount` EST_UN `BankAccount`."

La deuxième question est plus subtile. Quand il lui est passé un objet `BankAccount`, `MakeAWithdrawal()` invoque `BankAccount.Withdraw()`. C'est assez clair. Mais lorsqu'il lui est passé un objet `SavingsAccount`, `MakeAWithdrawal()` appelle la même méthode. Ne devrait-elle pas invoquer la méthode `Withdraw()` dans la sous-classe ?

Le procureur veut montrer que l'appel à `ba.Withdraw()` devrait invoquer la méthode `BankAccount.Withdraw()`. Il est clair que l'objet `ba` est un `BankAccount`. Faire autre chose ne pourrait que faire naître la confusion. Mais la défense a des témoins dans `Main()` pour prouver que bien que l'objet `ba` soit déclaré comme `BankAccount`, c'est en fait un `SavingsAccount`. Le jury ne s'y retrouve plus. Les deux arguments sont tout aussi valides l'un que l'autre.

Dans ce cas, C# se range du côté du procureur. Le choix le plus sûr est de s'en tenir au type déclaré, parce qu'il évite toute erreur de communication. L'objet est donc déclaré être un `BankAccount`, et la cause est entendue.

Qu'y a-t-il de mal à utiliser chaque fois le type déclaré ?

Dans certains cas, vous ne voudrez pas utiliser le type déclaré. Ce que vous voudrez vraiment, c'est effectuer l'appel sur la base du *type réel*, c'est-à-dire le type à l'exécution, par opposition au type déclaré. Cette possibilité de décider à l'exécution s'appelle *polymorphisme*, ou *late binding* (liaison tardive). Utiliser le type déclaré s'appelle *early binding* (liaison précoce).

```
public class Class1
{
  public static int Main(string[] args)
  {
    // crée un int et l'initialise à la valeur 0
    int i = new int();
    // lui assigne une valeur et la restitue par
    // l'interface IFormattable
    i = 1;
    OutputFunction(i);
    // la constante 2 implémente la même interface
    OutputFunction(2);
    // en fait, vous pouvez utiliser directement le même objet
    Console.WriteLine("Extrait directement = {0}", 3.ToString());
    // ceci peut être très utile ; vous pouvez extraire un int
    // d'une liste :
    Console.WriteLine("\nExtrait d'une liste les nombres entiers");
    object[] objects = new object[5];
    objects[0] = "this is a string";
    objects[1] = 2;
    objects[2] = new Class1();
    objects[3] = 4;
    objects[4] = 5.5;
    for(int index = 0; index < objects.Length; index++)
    {
      if (objects[index] is int)
      {
        int n = (int)objects[index];
        Console.WriteLine("L'élément numéro {0} est {1}",
                          index, n);
      }
    }
    // autre utilisation de l'unification des types
    Console.WriteLine("\nAffichage de tous les objets de la liste");
    int nCount = 0;
    foreach(object o in objects)
    {
      Console.WriteLine("Objets[{0}] est <{1}>",
        nCount++, o.ToString());
    }
    // attend confirmation de l'utilisateur
    Console.WriteLine("Appuyez sur Entrée pour terminer...");
    Console.Read();
    return 0;
  }
  // OutputFunction - affiche toute méthode qui implémente
  //                  ToString()
  public static void OutputFunction(IFormattable id)
  {
```

```
    BankAccount ba = (BankAccount)this;
    // invoque Withdraw() en utilisant cet objet BankAccount
    // appelle la fonction BankAccount.Withdraw()
    double dAmountWithdrawn = ba.Withdraw(dWithdrawal);
    if (++nNumberOfWithdrawalsThisPeriod > 1)
    {
      dAmountWithdrawn += ba.Withdraw(1.5);
    }
    return dAmountWithdrawn;
  }
```

Cette solution fonctionne : l'appel `ba.Withdraw()` invoque maintenant la méthode `BankAccount`, comme on le voulait. L'inconvénient de cette approche est la référence explicite à `BankAccount`. Une modification ultérieure du programme pourrait modifier la hiérarchie d'héritage de telle manière que `SavingsAccount` n'hérite plus directement de `BankAccount`. Une telle réorganisation brise cette fonction d'une façon qu'un nouveau programmeur pourra avoir du mal à trouver. Pour moi, je n'arriverais jamais à trouver un bogue comme celui-là.

Il vous faut un moyen de dire à C# d'appeler la fonction `Withdraw()` depuis "la classe qui est juste au-dessus" dans la hiérarchie, sans la nommer explicitement. Ce serait la classe qui est étendue par `SavingsAccount`. C'est dans ce but que C# comporte le mot-clé `base`.



C'est le même mot-clé `base` qu'utilise un constructeur pour passer des arguments au constructeur de la classe de base.

Le mot-clé `base` de C# est la même chose que `this`, mais redéfinit le cast à la classe de base, quelle que soit cette classe :

```
// Withdraw - tout retrait est autorisé jusqu'à la valeur
//            du solde ; retourne le montant retiré
new public decimal Withdraw(decimal mWithdrawal)
{
  // soustrait 1.50 F
  base.Withdraw(1.5M);
  // vous pouvez maintenant effectuer un retrait avec ce qui reste
  return base.Withdraw(mWithdrawal);
}
```

L'appel `base.Withdraw()` invoque maintenant la méthode `BankAccount.Withdraw()`, évitant par-là l'écueil qui consiste à s'invoquer elle-même. En outre, cette solution ne sera pas brisée si la hiérarchie d'héritage est modifiée.

L'appel suivant est l'appel à la fonction `ChangeReferenceFunction()`. Cette fonction apparaît identique à `ChangeValueFunction()`, à l'exception de l'ajout du mot-clé `ref` à la liste des arguments. Comme `test` est maintenant passé par référence, l'argument `t` se réfère à l'objet original `test` et non à une copie nouvellement créée.

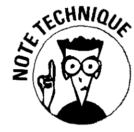
Le dernier appel de `Main()` est un appel à la méthode `ChangeMethod()`. Comme un appel à une méthode passe toujours l'objet courant par référence, les modifications effectuées par cette méthode sont conservées une fois de retour dans `Main()`.

La sortie de ce programme se présente de la façon suivante :

```
Valeur initiale de test
id = (10.00, 20.00)
Valeur de test après l'appel ChangeValueFunction(100, 200.0)
id = (10.00, 200.00)
Valeur de test après l'appel ChangeReferenceFunction(100, 200.0)
id = (100.00, 200.00)
Valeur de test après l'appel ChangeMethod(1000, 2000.0)
id = (1,000.00, 2,000.00)
Appuyez sur Entrée pour terminer...
```

Réconcilier la valeur et la référence : unifier le système de types

Les structures et les classes présentent une similitude frappante : toutes deux dérivent d'`Object`. En fait, toutes les classes et toutes les structures, qu'elles le disent ou non, dérivent d'`Object`. C'est ce qui unifie les différents types de variables.



Cette unification des types de variables est étrangère aux autres langages dérivés de C, comme C++ et Java. En fait, la séparation entre les objets de type référence et de type valeur en Java peut être un véritable casse-tête. Comme tout est un casse-tête en C++, un de plus ou de moins ne se remarque même pas.

Les types structure prédéfinis

La similitude entre les types structure et les types valeur simples n'est pas que superficielle. En fait, un type valeur simple est une structure. Par

Et si je redéfinis accidentellement une méthode de la classe de base ?

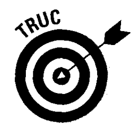
Il peut arriver à tout le monde de redéfinir accidentellement une méthode de la classe de base. Par exemple, je peux avoir une méthode `Véhicule.Virage()` qui fait tourner le véhicule. Plus tard, quelqu'un étend ma classe `Véhicule` avec une classe `Avion`, dont la méthode `Virage()` est entièrement différente. Il est clair que nous avons là un cas de confusion d'identité. Ces deux méthodes n'ont rien à voir l'une avec l'autre, sinon qu'elles portent le même nom.

Heureusement pour nous, C# sait détecter ce problème.

En compilant l'exemple précédent, `HidingWithdraw()`, C# génère un avertissement patibulaire. Le texte de ce message est un peu long, mais en voici la partie importante :

```
Le mot-clé new est requis sur
'HidingWithdrawal.SavingsAccount.Withdraw(decimal)', car il
masque le membre hérité
'HidingWithdrawal.BankAccount.Withdraw(decimal)'
```

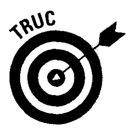
C# essaie de vous dire que vous avez écrit dans une sous-classe une méthode portant le même nom qu'une méthode de la classe de base. Est-ce vraiment ce que vous vouliez faire ?



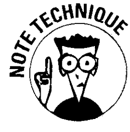
Ce message n'est qu'un avertissement. Vous ne le remarquerez même pas, à moins de passer à la fenêtre Sortie pour voir ce qui y est affiché. Dans presque tous les cas, vous y verrez un avertissement qui vous prévient que quelque chose pourrait bien vous mordre si vous n'y mettez pas bon ordre.

Le descripteur `new` indique à C# qu'une méthode est redéfinie intentionnellement et que ce n'est pas le résultat d'une négligence :

```
// plus de problèmes avec withdraw()
new public decimal Withdraw(decimal dWithdrawal)
{
  // . . . pas de modifications internes. . .
}
```



Cette utilisation du mot-clé `new` n'a rien à voir avec l'utilisation du même mot-clé pour créer un objet.



Je me permettrai de faire remarquer ici que c'est l'une des choses que je trouve agaçantes chez C# (et C++ avant lui) : faites ce que vous voulez avec mes méthodes, mais ne surchargez pas mes mots-clés. Quand je dis `new`, c'est que je veux créer un objet. Ils auraient pu utiliser un autre mot-clé pour indiquer une surcharge intentionnelle.

```
    // une struct peut avoir une méthode
    public void ChangeMethod(int nNewValue, double dNewValue)
    {
      n = nNewValue;
      d = dNewValue;
    }
    // ToString - redéfinit la méthode ToString dans l'objet
    override public string ToString()
    {
      return string.Format("({0:N}, {1:N})", n, d);
    }
  }
  public class Class1
  {
    public static int Main(string[] args)
    {
      // crée un objet Test
      Test test = new Test(10);
      Console.WriteLine("Valeur initiale de test");
      OutputFunction(test);
      // essaie de modifier l'objet de test en le passant
      // comme argument
      ChangeValueFunction(test, 100, 200.0);
      Console.WriteLine("Valeur de test après l'appel" +
                        " ChangeValueFunction(100, 200.0)");
      OutputFunction(test);
      // essaie de modifier l'objet de test en le passant
      // comme argument
      ChangeReferenceFunction(ref test, 100, 200.0);
      Console.WriteLine("Valeur de test après l'appel" +
                    " ChangeReferenceFunction(100, 200.0)");
      OutputFunction(test);
      // une méthode peut modifier l'objet
      test.ChangeMethod(1000, 2000.0);
      Console.WriteLine("Valeur de test après l'appel" +
                           " ChangeMethod(1000, 2000.0)");
      OutputFunction(test);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
    // ChangeValueFunction - passe la struct par référence
    public static void ChangeValueFunction(Test t,
                              int newValue, double dNewValue)
    {
      t.N = newValue;
      Test.D = dNewValue;
    }
```

```
    }
  }
  // SavingsAccount - compte bancaire qui rapporte des intérêts
  public class SavingsAccount : BankAccount
  {
    public decimal mInterestRate;
    // SavingsAccount - lit le taux d'intérêt, exprimé en
    //                  pourcentage (valeur comprise entre 0 et 100)
    public SavingsAccount(decimal mInitialBalance,
                          decimal mInterestRate)
    : base(mInitialBalance)
    {
      this.mInterestRate = mInterestRate / 100;
    }
    // AccumulateInterest - invoquée une fois par période
    public void AccumulateInterest()
    {
      mBalance = mBalance + (mBalance * mInterestRate);
    }
    // Withdraw - tout retrait est autorisé jusqu'à la valeur
    //            du solde ; retourne le montant retiré
    public decimal Withdraw(decimal mWithdrawal)
    {
      // soustrait 1.50 F
      base.Withdraw(1.5M);
      // vous pouvez maintenant effectuer un retrait avec ce qui reste
      return base.Withdraw(mWithdrawal);
    }
  }
  public class Class1
  {
    public static void MakeAWithdrawal(BankAccount ba,
                                       decimal mAmount)
    {
      ba.Withdraw(mAmount);
    }
    public static int Main(string[] args)
    {
      BankAccount ba;
      SavingsAccount sa;
      // crée un compte bancaire, en retire 100 F, et
      // affiche les résultats
      ba = new BankAccount(200M);
      ba.Withdraw(100M);
      // essaie de faire la même chose avec un compte rémunéré
      sa = new SavingsAccount(200M, 12);
      sa.Withdraw(100M);
      // affiche le solde résultant
```

Vous pouvez réaliser vous-même un constructeur (qui n'est donc pas un constructeur par défaut) qui fait effectivement quelque chose :

```
    public struct Test
    {
      private int n;
      public Test(int n)
      {
        this.n = n;
      }
  }
  public class Class1
  {
    public static void Main(string[] args)
    {
       Test test = new Test(10);
    }
  }
```

En dépit des apparences, la déclaration `test = new Test(10);` **n'alloue pas de mémoire**. Elle ne fait qu'initialiser la mémoire du type valeur qui est déjà là.

Les méthodes d'une structure sont rusées

Une structure peut avoir des membres qui en sont des instances, notamment des méthodes et des propriétés. Une structure peut avoir des membres statiques. Ces derniers peuvent avoir des initialiseurs, mais les membres non statiques (les instances) ne le peuvent pas. Normalement, un objet de type structure est passé à une fonction par valeur, mais il peut être passé par référence, à condition que ce soit spécifiquement indiqué dans l'appel à la fonction. Une structure ne peut pas hériter d'une classe (autre que `Object`, comme je l'explique dans la section "Réconcilier la valeur et la référence : unifier le système de types", plus loin dans ce chapitre), et une classe ne peut pas en hériter. Une structure peut implémenter une interface.



Si vous ne vous souvenez pas de la différence entre un membre statique et une instance, reportez-vous au Chapitre 8. Pour vous rafraîchir la mémoire sur le passage par valeur et le passage par référence, voyez le Chapitre 7. Le Chapitre 12 traite de l'héritage. Et si vous ne savez pas ce qu'est une interface, c'est dans ce chapitre que vous trouverez la réponse.

Redéfinir une méthode d'une classe de base

Ainsi, une méthode d'une classe peut surcharger une autre méthode de la même classe en ayant des arguments différents. De même, une méthode peut aussi surcharger une méthode de sa classe de base. Surcharger une méthode d'une classe de base s'appelle *redéfinir*, ou *cacher* la méthode.

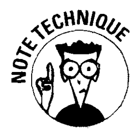
Imaginez que ma banque adopte une politique qui établisse une différence entre les retraits sur les comptes rémunérés et les autres types de retrait. Pour les besoins de notre exemple, imaginez aussi qu'un retrait effectué sur un compte rémunéré coûte une commission de 1,50 F.

Avec l'approche fonctionnelle, vous pourriez implémenter cette politique en définissant dans la classe un indicateur qui dise si l'objet est un `SavingsAccount` ou un simple `BankAccount`. La méthode de retrait devrait alors tester l'indicateur pour savoir si elle doit ou non imputer la commission de 1,50 F :

```
public BankAccount(int nAccountType)
{
  private decimal mBalance;
  private bool isSavingsAccount;
  // indique le solde initial et dit si le compte
  // que vous êtes en train de créer est ou non
  // un compte rémunéré
  public BankAccount(decimal mInitialBalance,
                     bool isSavingsAccount)
  {
    mBalance = mInitialBalance;
    this.isSavingsAccount = isSavingsAccount;
  }
  public decimal Withdraw(decimal mAmount)
  {
    // si le compte est un compte rémunéré . . .
    if (isSavingsAccount)
    {
      // ...alors soustrait 1.50 F
      mBalance -= 1.50M;
    }
    // poursuit avec le même code pour le retrait :
    if (mAmountToWithdraw > mBalance)
    {
      mAmountToWithdraw = mBalance;
    }
    mBalance -= mAmountToWithdraw;
    return mAmountToWithdraw;
```

```
n = 1;
// la déclaration d'une struct ressemble à la déclaration d'un simple int
MyStruct ms;
ms.n = 3;       // accède aux membres comme à un objet de classe
ms.d = 3.0;
// un objet de classe doit être alloué à partir
// d'une zone séparée de la mémoire
MyClass mc = new MyClass;
mc.n = 2;
mc.d = 2.0;
```

Un objet `struct` est stocké en mémoire de la même manière qu'une variable intrinsèque. La variable `ms` n'est pas une référence à un bloc de mémoire externe alloué à partir d'une zone de mémoire séparée.



La "zone de mémoire spéciale" dont viennent les objets de classe s'appelle le *tas* (the heap). Ne me demandez pas pourquoi.

L'objet `ms` se trouve dans la même zone de mémoire locale que la variable `n`, comme le montre la Figure 14.1.

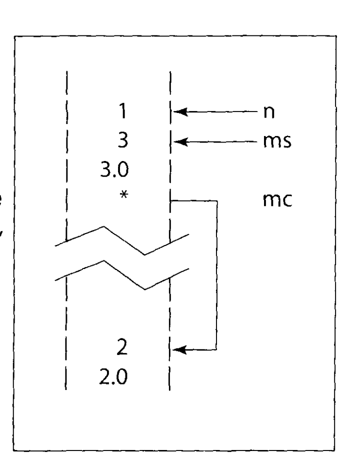


Figure 14.1 : La variable structure `ms` "réside" dans la même zone de mémoire que la variable de type valeur `n`, alors que la zone dans laquelle réside l'objet `mc` vient d'un espace mémoire particulier du tas.

La distinction entre un type référence et un type valeur est encore plus évidente dans l'exemple qui suit. L'allocation d'un tableau de 100 objets

De notre point de vue d'être humain, la différence entre un four à micro-ondes et un four conventionnel ne semble pas de la plus haute importance, mais envisagez un instant la question du point de vue du four. Les étapes du processus interne mis en œuvre par un four conventionnel sont complètement différentes de celles d'un four à micro-ondes (sans parler d'un four à convection).

Le pouvoir du principe de l'héritage repose sur le fait qu'une sous-classe n'est pas *obligée* d'hériter à l'identique de toutes les méthodes de la classe de base. Une sous-classe peut hériter de l'essence des méthodes de la classe de base tout en réalisant une implémentation différente de leurs détails.

Surcharger une méthode héritée

Plusieurs fonctions peuvent porter le même nom, à condition qu'elles soient différenciées par le nombre et/ou le type de leurs arguments.

Ce n'est qu'une question de surcharge de fonction



Donner le même nom à deux fonctions (ou plus) s'appelle *surcharger* un nom de fonction.

Les arguments d'une fonction font partie de son nom complet, comme le montre l'exemple suivant :

```
public class MyClass
{
  public static void AFunction()
  {
    // faire quelque chose
  }
  public static void AFunction(int)
  {
    // faire quelque chose d'autre
  }
  public static void AFunction(double d)
  {
    // faire encore quelque chose d'autre
  }
  public static void Main(string[] args)
  {
```

l'exigence d'implémenter CompareTo(). CompareTo() y ajoute GetValue(), qui retourne la valeur des objets dans un int.



Bien qu'elle puisse retourner la valeur de l'objet sous forme int, GetValue() ne dit rien sur ce que contient la classe. La génération d'une valeur int peut très bien mettre en jeu un calcul complexe.

La classe BaseClass implémente l'interface ICompare (la méthode concrète GetValue() retourne le membre donnée nValue). Toutefois, la méthode CompareTo(), qui est également exigée par l'interface ICompare, est déclarée abstract.



Déclarer une classe abstract signifie qu'il s'agit d'un concept incomplet, auquel manque l'implémentation d'une ou de plusieurs propriétés (dans ce cas, la méthode CompareTo()).

SubClass fournit la méthode CompareTo() nécessaire.

Remarquez que SubClass implémente automatiquement l'interface ICompare, bien qu'elle ne le dise pas explicitement. BaseClass avait promis d'implémenter les méthodes de Icompare, et SubClass EST_UNE BaseClass. En héritant de ses méthodes, SubClass implémente automatiquement Icompare.

Main() crée deux objets de la classe SubClass avec des valeurs différentes. Elle passe ensuite des objets à MyFunc(). La méthode MyFunc() s'attend à recevoir deux objets de l'interface ICompare. MyFunc() utilise la méthode CompareTo() pour décider quel objet est le plus grand, puis GetValue() pour afficher la "valeur" des deux objets.

Ce programme donne une sortie courte et agréable :

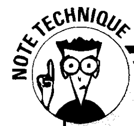
```
La valeur de ic1 est 10 et celle de ic2 est 20
Les objets eux-mêmes considèrent que ic1 est plus petit que ic2
Appuyez sur Entrée pour terminer...
```

Une structure n'a pas de classe

C# semble être doté d'une double personnalité pour la manière de déclarer les variables. Les variables d'un type valeur comme int et double sont déclarées et initialisées d'une certaine manière :

```
int n;
n = 1;
```

Dans le cas d'une succession d'héritages de classes, les destructeurs sont invoqués dans l'ordre inverse des constructeurs. Autrement dit, le destructeur de la sous-classe est invoqué avant le destructeur de la classe de base.



Le ramasse-miettes et le destructeur C#

La méthode du destructeur est beaucoup moins utile en C# que dans d'autres langages orientés objet, comme C++, car C# possède de ce l'on appelle une destruction non déterministe.

La mémoire allouée à un objet est supprimée du tas lorsque le programme exécute la commande `new`. Ce bloc de mémoire reste réservé aussi longtemps que les références valides à celui-ci restent actives.

Une zone de mémoire est dite "inaccessible" lorsque la dernière référence à celle-ci passe hors de portée. Autrement dit, personne ne peut plus accéder à cette zone de mémoire quand plus rien n'y fait référence.

C# ne fait rien de particulier lorsqu'une zone de mémoire devient inaccessible. Une tâche de faible priorité est exécutée à l'arrière-plan, recherchant les zones de mémoire inaccessibles. Ce qu'on appelle le ramasse-miettes s'exécute à un faible niveau de priorité afin d'éviter de diminuer les performances du programme. Le ramasse-miettes restitue au tas les zones de mémoire inaccessibles qu'il trouve.

En temps normal, le ramasse-miettes opère en silence à l'arrière-plan. Il ne prend le contrôle du programme qu'à de brefs moments, lorsque le tas est sur le point d'être à court de mémoire.

Le destructeur de C# est non déterministe parce qu'il ne peut pas être invoqué avant que l'objet ait été récupéré par le ramasse-miettes, ce qui peut se produire longtemps après qu'il a cessé d'être utilisé. En fait, si le programme se termine avant que l'objet soit trouvé par le ramasse-miettes et retourné au tas, le destructeur n'est pas invoqué du tout.

Au bout du compte, l'effet qui en résulte est qu'un programmeur C# ne peut pas se reposer sur le destructeur pour opérer automatiquement comme dans un langage comme C++.




```
// AbstractInterface - montre comment une interface
//                     peut être implémentée avec
//                     une classe abstraite
using System;
namespace AbstractInterface
{
  // ICompare - interface capable de se comparer elle-même
  //            et d'afficher sa propre valeur
  public interface ICompare : IComparable
  {
    // GetValue - retourne sa propre valeur sous forme d'un int
    int GetValue();
  }
  // BaseClass - implémente l'interface ICompare en
  //             fournissant une méthode concrète GetValue() et
  //             une méthode abstraite CompareTo()
  abstract public class BaseClass : ICompare
  {
    int nValue;
    public BaseClass(int nInitialValue)
    {
      nValue = nInitialValue;
    }
    // implémente d'abord l'interface ICompare
    // avec une méthode concrète
    public int GetValue()
    {
      return nValue;
    }
    // complète l'interface ICompare avec une méthode abstraite
    abstract public int CompareTo(object rightObject);
  }
  // SubClass - complète la classe de base en redéfinissant
  //            la méthode abstraite CompareTo()
  public class SubClass: BaseClass
  {
    // passe au constructeur de la classe de base
    // la valeur passée au constructeur précédent
    public SubClass(int nInitialValue) : base(nInitialValue)
    {
    }
    // CompareTo - implémente l'interface IComparable ; retourne
    //             une indication disant si un objet d'une sous-classe
    //             est plus grand qu'un autre
    override public int CompareTo(object rightObject)
    {
      BaseClass bc = (BaseClass)rightObject;
      return GetValue().CompareTo(bc.GetValue());
```

```
    {
      nAccountNumber = ++nNextAccountNumber;
      mBalance = mInitialBalance;
    }
    // . . . même chose ici . . .
  }
  // SavingsAccount - compte bancaire qui rapporte des intérêts
  public class SavingsAccount : BankAccount
  {
    public decimal mInterestRate;
    // InitSavingsAccount - lit le taux d'intérêt, exprimé en
    //                      pourcentage (valeur comprise entre 0 et 100)
    public SavingsAccount(decimal mInterestRate) : this(0, mInterestRate)
    {
    }
    public SavingsAccount(decimal mInitial,
                          decimal mInterestRate) : base(mInitial)
    {
      this.mInterestRate = mInterestRate / 100;
    }
    // . . . même chose ici . . .
  }
  public class Class1
  {
    // DirectDeposit - effectue automatiquement le dépôt d'un chèque
    public static void DirectDeposit(BankAccount ba,
                                     decimal mPay)
    {
      ba.Deposit(mPay);
    }
    public static int Main(string[] args)
    {
      // crée un compte bancaire et l'affiche
      BankAccount ba = new BankAccount(100);
      DirectDeposit(ba, 100);
      Console.WriteLine("Compte {0}", ba.ToBankAccountString());
      // et maintenant un compte rémunéré
      SavingsAccount sa = new SavingsAccount(12.5M);
      DirectDeposit(sa, 100);
      sa.AccumulateInterest();
      Console.WriteLine("Compte {0}", sa.ToSavingsAccountString());
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
}
```

Le tableau trié d'objets Student est alors passé à la méthode localement définie DisplayArray(). Celle-ci effectue une itération sur un tableau d'objets qui implémentent GetString(). Elle utilise la propriété Array.Length pour savoir combien d'objets contient le tableau, puis elle appelle GetString() pour chaque objet, et affiche le résultat sur la console en utilisant WriteLine().

Le programme, de retour dans Main(), continue en triant et en affichant les oiseaux. Je suppose que vous conviendrez que les oiseaux n'ont rien à voir avec des étudiants, mais la classe Bird implémente l'interface IComparable en comparant les noms des oiseaux, et l'interface IDisplayable en retournant le nom de chaque oiseau.

Remarquez que Main() ne récupère pas cette fois le tableau des oiseaux. Ce n'est pas nécessaire. C'est la même chose que ce fragment de code :

```
class BaseClass {}
class SubClass : BaseClass {}
class Class1
{
  public static void SomeFunction(BaseClass bc) {}
  public static void AnotherFunction()
  {
    SubClass sc = new SubClass();
    SomeFunction(sc);
  }
}
```

Ici, un objet de la classe SubClass peut être passé à la place d'un objet d'une classe de base, parce qu'une sous-classe EST_UNE classe de base.

De même, un tableau d'objets Bird peut être passé à une méthode attendant un tableau d'objets IComparable, parce que Bird implémente cette interface. L'appel suivant à DisplayArray() passe le tableau birds, encore une fois sans cast, parce que Bird implémente l'interface IDisplayable.

La sortie du programme se présente de la façon suivante :

```
Tri de la liste des étudiants
Lisa     :100
Marge    :85
Bart     :50
Maggie   :30
Homer    :0
```

```
{
  public class BaseClass
  {
    public BaseClass()
    {
      Console.WriteLine("Construction de BaseClass (default)");
    }
    public BaseClass(int i)
    {
      Console.WriteLine("Construction de BaseClass({0})", i);
    }
  }
  public class SubClass : BaseClass
  {
    public SubClass()
    {
      Console.WriteLine("Construction de SubClass (default)");
    }
    public SubClass(int i1, int i2) : base(i1)
    {
      Console.WriteLine("Construction de SubClass({0}, {1})",
                                              i1,  i2);
    }
  }
  public class Class1
  {
    public static int Main(string[] args)
    {
      Console.WriteLine("Invocation de SubClass()");
      SubClass sc1 = new SubClass();
      Console.WriteLine("\nInvocation de SubClass(1, 2)");
      SubClass sc2 = new SubClass(1, 2);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
}
```

Ce programme donne la sortie suivante :

```
Invocation de SubClass()
Construction de BaseClass (default)
Construction de SubClass (default)

Invocation de SubClass(1, 2)
```

```
  public string GetString()
  {
    string sPadName = Name.PadRight(9);
    string s = String.Format("{0}:{1:N0}",
                             sPadName, Grade);
    return s;
  }
}
// -----------Birds - tri des oiseaux par nom --------
// Bird - tableau de noms d'oiseau
class Bird : IComparable, IDisplayable
{
  private string sName;
  // Constructor - initialise un nouvel objet student
  public Bird(string sName)
  {
    this.sName = sName;
  }
  // CreateBirdList - retourne une liste d'oiseaux à la fonction appelante
  static string[] sBirdNames =
     { "Tourterelle", "Vautour", "Alouette", "Étourneau",
       "Grive", "Corbeau", "Hirondelle"};
  public static Bird[] CreateBirdList()
  {
    Bird[] birds = new Bird[sBirdNames.Length];
    for(int i = 0; i < birds.Length; i++)
    {
      birds[i] = new Bird(sBirdNames[i]);
    }
    return birds;
  }
  // accède aux méthodes en lecture seule
  public string Name
  {
    get
    {
      return sName;
    }
  }
  // implémente l'interface IComparable :
  // CompareTo - compare les noms des oiseaux, utilise
  //             la méthode de comparaison intégrée de la classe String
  public int CompareTo(object rightObject)
  {
    // nous allons comparer l'oiseau "courant" à
    // l'objet oiseau "de droite"
    Bird leftBird = this;
    Bird rightBird = (Bird)rightObject;
    return String.Compare(leftBird.Name, rightBird.Name);
```

dont elle réalise une extension. Il y a une raison à cela : chaque classe est responsable de ce qu'elle fait. Une sous-classe ne doit pas plus être tenue pour responsable de l'initialisation des membres de la classe de base qu'une fonction extérieure quelconque. La classe `BaseClass` doit se voir donner la possibilité de construire ses membres avant que les membres de `SubClass` aient la possibilité d'y accéder.

Passer des arguments au constructeur de la classe de base : le mot-clé `base`

La sous-classe invoque le constructeur par défaut de sa classe de base, sauf indication contraire, même à partir d'un constructeur d'une sous-classe autre que le constructeur par défaut. C'est ce que montre l'exemple légèrement modifié ci-dessous :

```
using System
namespace Example
{
  public class Class1
  {
    public static int Main(string[] args)
    {
      Console.WriteLine("Invocation de SubClass()");
      SubClass sc1 = new SubClass();
      Console.WriteLine("\nInvocation de SubClass(int)");
      SubClass sc2 = new SubClass(0);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
  }
  public class BaseClass
  {
    public BaseClass()
    {
      Console.WriteLine("Construction de BaseClass (default)");
    }
    public BaseClass(int i)
    {
      Console.WriteLine("Construction de BaseClass (int)");
    }
  }
  public class SubClass : BaseClass
```

```
      // explicite des objets...
      Array.Sort(birds);
      DisplayArray(birds);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
    // DisplayArray - affiche un tableau d'objets qui
    //                implémentent l'interface IDisplayable
    public static void DisplayArray
      (IDisplayable[] displayables)
    {
      int length = displayables.Length;
      for(int index = 0; index < length; index++)
      {
        IDisplayable displayable = displayables[index];
        Console.WriteLine("{0}", displayable.GetString());
      }
    }
  }
  // ---- Students - trie les étudiants par moyenne de points d'UV ----
  // Student - description d'un étudiant avec son nom et ses points d'UV
  class Student : IComparable, IDisplayable
  {
    private string sName;
    private double dGrade = 0.0;
    // Constructor - initialise un nouvel objet student
    public Student(string sName, double dGrade)
    {
      // met de côté les données de l'objet
      this.sName = sName;
      this.dGrade = dGrade;
    }
    // CreateStudentList - pour gagner de la place, crée
    //              une liste fixe d'étudiants
    static string[] sNames =
          {"Homer", "Marge", "Bart", "Lisa", "Maggie"};
    static double[] dGrades =
          {0, 85, 50, 100, 30};
    public static Student[] CreateStudentList()
    {
      Student[] sArray = new Student[sNames.Length];
      for (int i = 0; i < sNames.Length; i++)
      {
        sArray[i] = new Student(sNames[i], dGrades[i]);
      }
      return sArray;
    }
    // accède aux méthodes en lecture seule
```

```
    public static void GenericFunction(object o)
    {
      if (o is MyClass1)
      {
        MyClass1 mc1 = (MyClass1)o;
        // . . .
      }
    }
  }
```

GenericFunction() peut être invoquée avec n'importe quel type d'objet. Le mot-clé is extraira des huîtres object toutes les perles de MyClass1.

L'héritage et le constructeur

Le programme InheritanceExample que nous avons vu plus haut dans ce chapitre repose sur ces horribles fonctions Init... pour initialiser les objets BankAccount et SavingsAccount en leur donnant un état valide. Équiper ces classes de constructeurs est certainement la meilleure manière de procéder, mais elle introduit une petite complication.

Invoquer le constructeur par défaut de la classe de base

Le constructeur par défaut de la classe de base est invoqué chaque fois qu'une sous-classe est construite. Le constructeur de la sous-classe invoque automatiquement le constructeur de la classe de base, comme le montre cet exemple simple :




```
// InheritingAConstructor - montre que le constructeur
//                          de la classe de base est invoqué
//                          automatiquement
using System;
namespace InheritingAConstructor
{
  public class Class1
  {
```

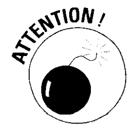
Cette méthode trie un tableau d'objets qui implémentent l'interface IComparable. La classe de ces objets n'a même pas d'importance. Ils pourraient très bien, par exemple, être des objets Student. La classe Array pourrait même trier la version suivante de Student :

```
// Student - description d'un étudiant avec son nom et ses points d'UV
class Student : IComparable
{
  private double dGrade;
  // accède aux méthodes en lecture seule
  public double Grade
  {
    get
    {
      return dGrade;
    }
  }
  // CompareTo - compare un étudiant à un autre ;
  //            un étudiant est "meilleur" qu'un autre
  //            si ses points d'UV sont meilleurs
  public int CompareTo(object rightObject)
  {
    Student leftStudent = this;
    Student rightStudent = (Student)rightObject;
    // génère maintenant -1, 0, ou 1 sur la base du
    // critère de tri (la moyenne des points d'UV de l'étudiant)
    if (rightStudent.Grade < leftStudent.Grade)
    {
      return -1;
    }
    if (rightStudent.Grade > leftStudent.Grade)
    {
      return 1;
    }
    return 0;
  }
}
```

Le tri d'un tableau d'objets Student est réduit à un simple appel :

```
void MyFunction(Student[] students)
{
  // trie le tableau d'objets IComparable
  Array.Sort(students);
}
```

Vous fournissez le comparateur, et Array fait tout le travail.



Une conversion incorrecte génère une erreur à l'exécution du programme (ce qu'on appelle une erreur *run-time*). Une erreur à l'exécution est beaucoup plus difficile à identifier et corriger qu'une erreur à la compilation.

Éviter les conversions invalides en utilisant le mot-clé `is`

La fonction `ProcessAmount()` se porterait très bien si elle pouvait être sûre que l'objet qui lui est passé est bien un `SavingsAccount` avant d'effectuer la conversion. C'est dans ce but que C# offre le mot-clé `is`.

L'opérateur `is` admet un objet à sa gauche et un type à sa droite. Il retourne `true` si le type à l'exécution de l'objet qui est à sa gauche est compatible avec le type qui est à sa droite.

Vous pouvez modifier l'exemple précédent pour éviter l'erreur à l'exécution en utilisant l'opérateur `is` :

```
public static void ProcessAmount(BankAccount bankAccount)
{
  // dépose une grosse somme sur le compte
  bankAccount.Deposit(10000.00);
  // si l'objet est un SavingsAccount . . .
  if (bankAccount is SavingsAccount)
  {
    // ...recueille l'intérêt dès maintenant
    SavingsAccount SavingsAccount = (SavingsAccount)bankAccount;
    savingsAccount.AccumulateInterest();
  }
}
public static void TestCast()
{
  SavingsAccount sa = new SavingsAccount();
  ProcessAmount(sa);
  BankAccount ba = new BankAccount();
  ProcessAmount(ba);
}
```

L'instruction `if` supplémentaire teste l'objet `bankAccount` pour vérifier qu'il est bien de la classe `SavingsAccount`. L'opérateur `is` retourne `true` lorsque `ProcessAmount()` est appelée pour la première fois. Toutefois, lorsqu'un objet `bankAccount` lui est passé dans le deuxième appel, l'opérateur `is` retourne `false`, évitant ainsi le cast invalide. Cette version de ce programme ne génère pas d'erreur à l'exécution.

```
      get
      {
        return dGrade;
      }
    }
    // GetString - retourne une représentation de l'étudiant
    public string GetString()
    {
      string sPadName = Name.PadRight(9);
      string s = String.Format("{0}:{1:N0}",
                               sPadName, Grade);
      return s;
    }
  }
```

L'appel à `PadRight()` garantit que le nom dans lequel sera inséré le champ aura au moins neuf caractères de long. Si le nombre fait moins de neuf caractères, la différence est comblée par des espaces. Ajuster une chaîne à une longueur standard permet d'aligner des objets en colonne. L'indication `{1:N0}` dit, "afficher le grade avec des virgules ou des points (selon le paramètre de localisation) comme séparateur de milliers." L'indication {0} qui précède arrondit la partie décimale.

Avec cette déclaration, je peux maintenant écrire le fragment de programme suivant (le programme complet est donné dans la section "Assembler le tout", plus loin dans ce chapitre) :

```
  // DisplayArray - affiche un tableau d'objets qui
  //                implémentent l'interface IDisplayable
  public static void DisplayArray
                 (IDisplayable[] displayables)
  {
    int length = displayables.Length;
    for(int index = 0; index < length; index++)
    {
      IDisplayable displayable = displayables[index];
      Console.WriteLine("{0}", displayable.GetString());
    }
  }
```

Cette méthode `DisplayArray()` peut afficher n'importe quel type de tableau, à condition que les membres du tableau définissent une méthode `GetString()`. Voici un exemple de sortie de `DisplayArray()` :

```
Homer    :0
Marge    :85
```

de façon ambiguë, faire démarrer une voiture n'est pas la même chose que faire démarrer un moteur. L'opération démarrage de la voiture dépend évidemment du démarrage du moteur, mais ce sont deux choses distinctes : il faut aussi passer la première, lâcher les freins, et ainsi de suite.

Plus encore, sans doute, faire hériter Car de Motor est une représentation erronée des choses. Une voiture n'est tout simplement pas un type particulier de moteur.



L'élégance du logiciel est un but qui se passe de justification. Non seulement elle le rend plus compréhensible, plus fiable et aisé à maintenir, mais elle réjouit le goût et facilite la digestion, entre autres.

Autres considérations

C# implémente un ensemble de caractéristiques conçues pour supporter l'héritage.

Changer de classe

Un programme peut changer la classe d'un objet. En fait, c'est une chose que vous avez déjà vue dans cet exemple. SomeFunction() peut passer un objet SavingsAccount à une méthode qui attend un objet BankAccount.

Vous pouvez rendre cette conversion plus explicite :

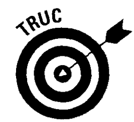
```
BankAccount ba;
SavingsAccount sa = new SavingsAccout();
                      // OK:
ba = sa;              // une conversion vers le bas implicite est admise
ba = (BankAccount)sa; // le cast explicite est préféré
                      // Non!
sa = ba;              // la conversion vers le haut implicite est interdite
                      // ceci est correct
sa = (SavingsAccount)ba;
```

La première ligne stocke un objet SavingsAccount dans une variable BankAccount. C# effectue pour vous cette conversion. La deuxième ligne utilise l'opérateur cast pour convertir explicitement l'objet.

Les deux dernières lignes reconvertissent l'objet BankAccount en SavingsAccount.

```
    // . . . écrire une note sur le PDA . . .
  }
}
public class Laptop : Computer, IRecordable
{
  public void TakeANote(string sNote)
  {
    // . . . taper une note sur le clavier . . .
  }
}
```

Chacune de ces trois classes hérite d'une classe de base différente, mais implémente la même interface IRecordable.



Notez la distinction dans la terminologie. On *hérite* d'une case de base, ou on l'*étend*, mais on *implémente* une interface. Ne me regardez pas comme ça. Je ne sais pas pourquoi ce sont ces termes qui ont été choisis, mais cette terminologie aide effectivement à faire les distinctions nécessaires.

L'interface IRecordable indique que chacune des trois classes peut être utilisée pour écrire une note en utilisant la méthode TakeANote(). Pour comprendre l'utilité de ce procédé, voyez la fonction suivante :

```
public class Class1
{
  static public void RecordShoppingList(IRecordable recordObject)
  {
    // créer une liste de commissions
    string sList = GenerateShoppingList();
    // puis la noter
    recordObject.TakeANote(sList);
  }
  public static void Main(string[] args)
  {
    PDA pda = new PDA();
    RecordShoppingList(pda);
  }
```

Concrètement, ce fragment de code dit que la fonction RecordShoppingList() accepte comme argument tout objet qui implémente la méthode TakeANote() (en termes humains, "tout objet qui peut enregistrer une note"). RecordShoppingList() ne fait aucune hypothèse sur le type exact d'objet recordObject. Que l'objet soit effectivement un PDA ou un certain type de ElectronicDevice est sans importance, pourvu qu'il puisse prendre une note.

```
  //                      pourcentage (valeur comprise entre 0 et 100)
  public void InitSavingsAccount(BankAccount bankAccount,
                                 decimal mInterestRate)
  {
    this.bankAccount = bankAccount;
    this.mInterestRate = mInterestRate / 100;
  }
  // AccumulateInterest - invoquée une fois par période
  public void AccumulateInterest()
  {
    bankAccount.mBalance = bankAccount.mBalance
                + (bankAccount.mBalance * mInterestRate);
  }
  // Deposit - tout dépôt positif est autorisé
  public void Deposit(decimal mAmount)
  {
    bankAccount.Deposit(mAmount);
  }
  // Withdraw - tout retrait est autorisé jusqu'à la valeur
  //           du solde ; retourne le montant retiré
  public double Withdraw(decimal mWithdrawal)
  {
    return bankAccount.Withdraw(mWithdrawal);
  }
}
```

Ici, la classe `SavingsAccount_` contient un membre donnée `bankAccount` (au lieu d'en hériter de `BankAccount`). L'objet `bankAccount` contient le solde et le numéro du compte, informations nécessaires pour la gestion du compte rémunéré. Les données propres à un compte rémunéré sont contenues dans la classe `SavingsAccount_`.

Dans ce cas, nous disons que `SavingsAccount_` A_UN `BankAccount`.

La relation A_UN

La relation A_UN est fondamentalement différente de la relation EST_UN. Cette différence ne semble pas mauvaise dans l'exemple de code suivant :

```
// crée un nouveau compte rémunéré
BankAccount ba = new BankAccount()
SavingsAccount_ sa = new SavingsAccount_();
sa.InitSavingsAccount(ba, 5);
// et y dépose cent euros
sa.Deposit(100);
```

Toutefois, cette solution souffre de deux gros problèmes. Le premier est fondamental : on ne peut pas prétendre que le stylo, le PDA et l'ordinateur portable sont liés par une relation quelconque de type EST_UN. Savoir comment fonctionne un stylo et comment s'en servir pour prendre une note ne me donne aucune information sur ce qu'est un ordinateur portable et la manière dont il enregistre les informations. Le nom `ThingsThatRecord` est plus une description qu'une classe de base.

Le second problème est purement technique. Il serait mieux de décrire `Laptop` comme une sous-classe de `Computer`. Bien que l'on puisse raisonnablement étendre la classe `PDA` à partir de la même classe de base `Computer`, on ne peut pas en dire autant de `Pen`. Il faudrait pour cela définir un stylo comme un certain type de `MechanicalWritingDevice` ou de `DeviceThatStainsYourShirt`. Toutefois, une classe C# ne peut pas hériter de deux classes différentes en même temps, elle ne peut être qu'un seul et même type de chose.

En revenant à nos trois classes initiales, le seul point que les classes `Pen`, `PDA` et `Laptop` ont en commun pour ce que nous voulons faire est qu'elles peuvent toutes être utilisées pour stocker quelque chose. La relation PEUT_ÊTRE_UTILISÉ_COMME `Recordable` nous permet de communiquer l'usage qui peut en être fait dans un but particulier, sans pour autant impliquer une relation inhérente entre ces trois classes.

Qu'est-ce qu'une interface ?

Une description d'interface ressemble beaucoup à une classe sans données, dans laquelle toutes les méthodes seraient abstraites. Une description d'interface pour des "choses qui enregistrent" pourrait ressembler à ceci :

```
interface IRecordable
{
  void TakeANote(String sNote);
}
```

Remarquer le mot-clé `interface` à la place de `class`. Entre les accolades se trouve une liste de méthodes abstraites. Une interface ne contient aucune définition de membre donnée.

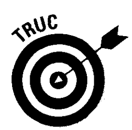
La méthode `TakeANote()` est écrite sans implémentation. Les mots-clés `public` et `virtual` ou `abstract` ne sont pas nécessaires. Toutes les méthodes d'une interface sont publiques, et une interface n'entre pas en jeu dans un héritage normal.

La propriété Balance permet de lire le solde, mais sans donner la possibilité de le modifier. La méthode Deposit() accepte tout dépôt positif. La méthode Withdraw() vous permet de retirer tout ce que vous voulez dans la limite de ce que vous avez sur votre compte. ToBankAccountString() crée une chaîne qui donne la description du compte.

La classe SavingsAccount hérite de toutes ces bonnes choses de BankAccount. À cela, elle ajoute un taux d'intérêt, et la possibilité d'accumuler des intérêts à intervalle régulier.

Main() en fait le moins possible. Elle crée un BankAccount, affiche le compte, crée un SavingsAccount, ajoute une période d'intérêts, et affiche le résultat :

```
Compte 1001 - 200,00 ¤
Compte 1002 - 112,50 ¤ (12,5%)
Appuyez sur Entrée pour terminer...
```



Remarquez que la méthode InitSavingsAccount() invoque InitBankAccount(). Cela initialise les membres donnée propres au compte. La méthode InitSavingsAccount() aurait pu les initialiser directement, mais il est de meilleure pratique de permettre à BankAccount d'initialiser ses propres membres.

EST_UN par rapport à A_UN – j'ai du mal à m'y retrouver

La relation entre SavingsAccount et BankAccount n'est rien d'autre que la relation fondamentale EST_UN. Pour commencer, je vais vous montrer pourquoi, puis je vous montrerai à quoi ressemblerait une relation A_UN.

La relation EST_UN

La relation EST_UN entre SavingsAccount et BankAccount est mise en évidence par la modification suivante à Class1 dans le programme SimpleSavingsAccount de la section précédente :

```
public class Class1
{
  // DirectDeposit - effectue automatiquement le dépôt d'un chèque
  public static void DirectDeposit(BankAccount ba,
```

Chapitre 14

Quand une classe n'est pas une classe : l'interface et la structure

Dans ce chapitre :

- Explorer la relation PEUT_ÊTRE_UTILISÉ_COMME.
- Définir une interface.
- Utiliser l'interface pour effectuer des opérations communes.
- Définir une structure.
- Utiliser la structure pour rassembler des classes, des interfaces et des types valeur intrinsèques.

Une classe peut contenir une référence à une autre classe. C'est alors une simple relation A_UN. Une classe peut étendre une autre classe par le merveilleux procédé de l'héritage. C'est une relation EST_UN. L'interface de C# implémente également une autre association, tout aussi importante : la relation PEUT_ÊTRE_UTILISÉ_COMME.

Qu'est-ce que PEUT_ÊTRE_UTILISÉ_COMME ?

Si je veux prendre en vitesse une petite note, je peux la griffonner sur un bout de papier avec un stylo, faire la même chose sur mon assistant numérique personnel (PDA) ou la taper sur mon ordinateur portable.

```
  //       (qui est égal à zéro par défaut)
  public void InitBankAccount()
  {
    InitBankAccount(0);
  }
  public void InitBankAccount(decimal mInitialBalance)
  {
    nAccountNumber = ++nNextAccountNumber;
    mBalance = mInitialBalance;
  }
  // Balance (solde)
  public decimal Balance
  {
    get { return mBalance;}
  }
  // Deposit - tout dépôt positif est autorisé
  public void Deposit(decimal mAmount)
  {
    if (mAmount > 0)
    {
      mBalance += mAmount;
    }
  }
  // Withdraw - tout retrait est autorisé jusqu'à la valeur
  //            du solde ; retourne le montant retiré
  public decimal Withdraw(decimal mWithdrawal)
  {
    if (mBalance <= mWithdrawal)
    {
      mWithdrawal = mBalance;
    }
    mBalance -= mWithdrawal;
    return mWithdrawal;
  }
  // ToString - met le compte sous forme de chaîne
  public string ToBankAccountString()
  {
    return String.Format("{0} - {1:C}",
      nAccountNumber, mBalance);
  }
}
// SavingsAccount - compte bancaire qui rapporte des intérêts
public class SavingsAccount : BankAccount
{
  public decimal mInterestRate;
  // InitSavingsAccount - lit le taux d'intérêt, exprimé en
  //                      pourcentage (valeur comprise entre 0 et 100)
  public void InitSavingsAccount(decimal mInterestRate)
```

Sceller une classe

Vous pouvez très bien décider que vous ne voulez pas que les générations futures de programmeurs puissent étendre une de vos classes. Dans ce cas, vous pouvez la verrouiller en utilisant le mot-clé sealed. Une classe scellée ne peut être utilisée comme classe de base pour une autre classe, quelle qu'elle soit.

Examinez le bloc de code suivant :

```
using System;
public class BankAccount
{
  // Withdrawal - tout retrait est autorisé jusqu'à la valeur
  //              du solde ; retourne le montant retiré
  virtual public void Withdraw(double dWithdraw)
  {
    Console.WriteLine("invoque BankAccount.Withdraw()");
  }
}
public sealed class SavingsAccount : BankAccount
{
  override public void Withdraw(double dWithdrawal)
  {
    Console.WriteLine("invoque SavingsAccount.Withdraw()");
  }
}
public class SpecialSaleAccount : SavingsAccount
{
  override public void Withdraw(double dWithdrawal)
  {
    Console.WriteLine("invoque SpecialSaleAccount.Withdraw()");
  }
}
```

Ce fragment de code produit l'erreur de compilation suivante :

```
'SpecialSaleAccount' : ne peut pas hériter de la classe
scellée 'SavingsAccount'
```

Le mot-clé sealed vous permet de protéger votre classe des assauts d'une éventuelle sous-classe. Par exemple, permettre aux programmeurs d'étendre une classe qui implémente la sécurité d'un système permettrait à celui qui le voudrait d'y introduire une porte dérobée.

Remarquez que la propriété EST_UN n'est pas réflexive : un `Student` EST_UNE `Person`, mais l'inverse n'est pas vrai. Une `Person` N'EST_PAS_UN `Student`. Un énoncé comme celui-ci se réfère toujours au cas général. Il pourrait se trouver qu'une `Person` particulière soit effectivement un `Student`, mais beaucoup de gens qui sont membres de la classe `Person` ne sont pas membres de la classe `Student`. En outre, la classe `Student` possède des propriétés qu'elle ne partage pas avec la classe `Person`. Par exemple, un `Student` a une moyenne de points d'UV, mais une `Person` ordinaire n'en a pas.

L'héritage est une propriété transitive. Par exemple, si je définis une nouvelle classe `GraduateStudent` comme une sous-classe de `Student`, alors un `GraduateStudent` est aussi une `Person`. Et il doit en être ainsi : si un `GraduateStudent` EST_UN `Student` et un `Student` EST_UNE `Person`, alors un `GraduateStudent` EST_UNE `Person`. CQFD.

À quoi me sert l'héritage ?

L'héritage a plusieurs fonctions importantes. Vous pourriez penser qu'il sert à réduire le volume de ce que vous avez à taper au clavier. Dans une certaine mesure, c'est vrai : lorsque je décris un objet de la classe `Student`, je n'ai pas besoin de répéter les propriétés d'une `Person`. Un aspect plus important, mais lié à celui-ci, est le grand mot d'ordre *réutiliser*. Les théoriciens des langages de programmation savent depuis longtemps qu'il est absurde de recommencer de zéro pour chaque nouveau projet en reconstruisant chaque fois les mêmes composants.

Comparez la situation du développement de logiciel à celle d'autres industries. Y a-t-il beaucoup de constructeurs automobile qui commencent par concevoir et fabriquer leurs propres pinces et tournevis pour construire une voiture ? Et même s'ils le faisaient, combien recommenceraient de zéro en réalisant des outils entièrement nouveaux pour chaque nouveau modèle ? Dans les autres industries, on s'est rendu compte qu'il est plus pertinent d'utiliser des vis et des écrous standards, et même des composants plus importants comme des moteurs, que de repartir de zéro chaque fois.

L'héritage permet de tirer le meilleur parti des composants logiciels existants. Vous pouvez adapter des classes existantes à de nouvelles applications sans leur apporter de modifications internes. C'est une nouvelle sous-classe, contenant les ajouts et les modifications nécessaires, qui hérite des propriétés d'une classe existante.

```
    {
      Console.WriteLine("    appelle SpecialSaleAccount.Withdraw()");
    }
  }

  // SaleSpecialCustomer - compte utilisé pour des clients particuliers
  //                       pendant la période des soldes
  public class SaleSpecialCustomer : SpecialSaleAccount
  {
    override public void Withdraw(double dWithdrawal)
    {
      Console.WriteLine
              ("    appelle SaleSpecialCustomer.Withdraw()");
    }
  }
}  }
}
```

Chacune de ces classes étend la classe qui se trouve au-dessus. Remarquez toutefois que `SpecialSaleAccount.Withdraw()` a été marquée comme `virtual`, brisant en ce point la chaîne des héritages. Dans la perspective de `BankAccount`, les classes `SpecialSaleAccount` et `SaleSpecialCustomer` ressemblent exactement à `SavingsAccount`. Ce n'est que dans la perspective d'un `SpecialSaleAccount` que la nouvelle version de `Withdraw()` devient disponible.

Cela est démontré par ce petit programme. La fonction `Main()` invoque une série de méthodes `Test()`, dont chacune est conçue pour accepter une sous-classe différente. Chacune de ces versions de `Test()` appelle `Withdraw()` dans la perspective d'un objet de classe différent.

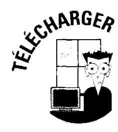
La sortie de ce programme se présente de la façon suivante :

```
Passage d'un BankAccount
  pour effectuer Test(BankAccount)
    appelle BankAccount.Withdraw()
Passage d'un SavingsAccount
  pour effectuer Test(BankAccount)
    appelle SavingsAccount.Withdraw()
  pour effectuer Test(SavingsAccount)
    appelle SavingsAccount.Withdraw()
Passage d'un SpecialSaleAccount
  pour effectuer Test(BankAccount)
    appelle SavingsAccount.Withdraw()
  pour effectuer Test(SavingsAccount)
```

Les langages orientés objet expriment cette relation d'héritage en permettant à une classe d'hériter d'une autre. C'est cette caractéristique qui permet aux langages orientés objet de produire des modèles plus proches du monde réel que les langages qui ne disposent pas du principe de l'héritage.

Hériter d'une classe

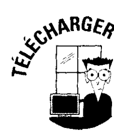
Dans l'exemple `InheritanceExample` suivant, la classe `SubClass` hérite de la classe `BaseClass` :




```
// InheritanceExample - offre la démonstration
//                      la plus simple de l'héritage
using System;
namespace InheritanceExample
{
  public class BaseClass
  {
    public int nDataMember;
    public void SomeMethod()
    {
      Console.WriteLine("SomeMethod()");
    }
  }
  public class SubClass : BaseClass
  {
    public void SomeOtherMethod()
    {
      Console.WriteLine("SomeOtherMethod()");
    }
  }
  public class Test
  {
    public static int Main(string[] args)
    {
      // crée un objet de la classe de base
      Console.WriteLine("Utilisons un objet de la classe de base :");
      BaseClass bc = new BaseClass();
      bc.nDataMember = 1;
      bc.SomeMethod();
      // créons maintenant un élément d'une sous-classe
      Console.WriteLine("Utilisons un objet d'une sous-classe :");
      SubClass sc = new SubClass();
      sc.nDataMember = 2;
      sc.SomeMethod();
```

Redémarrer une hiérarchie de classes

Le mot-clé `virtual` peut aussi être utilisé pour démarrer une nouvelle hiérarchie d'héritage. Examinez la hiérarchie de classes montrée dans le programme `InheritanceTest` :




```
// InheritanceTest - examine comment le mot-clé virtual
//                   peut être utilisé pour lancer
//                   une nouvelle hiérarchie d'héritage
namespace InheritanceTest
{
  using System;
  public class Class1
  {
    public static int Main(string[] strings)
    {
      Console.WriteLine("\nPassage d'un BankAccount");
      BankAccount ba = new BankAccount();
      Test1(ba);

      Console.WriteLine("\nPassage d'un SavingsAccount");
      SavingsAccount sa = new SavingsAccount();
      Test1(sa);
      Test2(sa);

      Console.WriteLine("\nPassage d'un SpecialSaleAccount");
      SpecialSaleAccount ssa = new SpecialSaleAccount();
      Test1(ssa);
      Test2(ssa);
      Test3(ssa);

      Console.WriteLine("\nPassage d'un SaleSpecialCustomer");
      SaleSpecialCustomer ssc = new SaleSpecialCustomer();
      Test1(ssc);
      Test2(ssc);
      Test3(ssc);
      Test4(ssc);

      // attend confirmation de l'utilisateur
      Console.WriteLine();
      Console.WriteLine("Hit Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }

    public static void Test1(BankAccount account)
```

Être avare de ses objets

On ne peut pas construire un objet sans un constructeur correspondant. Si vous définissez votre propre constructeur, C# retire le sien. En combinant ces deux aspects, vous pouvez créer une classe qui ne peut être instanciée que localement.

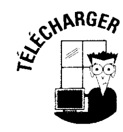
Par exemple, seule une méthode définie dans le même espace de nom que `BankAccount` peut créer un objet `BankAccount` avec le constructeur déclaré comme `internal` (pour en savoir plus sur les espaces de nom, reportez-vous au Chapitre 16) :

```
// BankAccount - simule un simple compte bancaire
public class BankAccount
{
  // les numéros de compte commencent à 1000 et augmentent
  // séquentiellement à partir de là
  static int nNextAccountNumber = 1000;
  // tient à jour le numéro de compte et le solde
  int nAccountNumber;
  double dBalance;
  // invoque le constructeur spécifique en fournissant
  // des valeurs par défaut pour les arguments manquants
  internal BankAccount()
  {
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0;
  }
  public string GetString()
  {
    return String.Format("#{0} = {1:N}",
                         nAccountNumber, dBalance);
  }
}
```

```
    // Output - classe abstraite qui affiche une chaîne
    abstract public void Output(string sOutputString);
  }
  // SubClass1 - implémentation concrète de AbstractBaseClass
  public class SubClass1 : AbstractBaseClass
  {
    override public void Output(string sSource)
    {
      string s = sSource.ToUpper();
      Console.WriteLine("Appel à SubClass1.Output() depuis {0}", s);
    }
  }
  // SubClass2 - autre implémentation concrète de AbstractBaseClass
  public class SubClass2 : AbstractBaseClass
  {
    override public void Output(string sSource)
    {
      string s = sSource.ToLower();
      Console.WriteLine("Appel à SubClass2.Output() depuis {0}", s);
    }
  }
  class Class1
  {
    public static void Test(AbstractBaseClass ba)
    {
      ba.Output("Test");
    }
    public static void Main(string[] strings)
    {
      /*
       * On ne peut pas créer d'objet AbstractBaseClass car c'est une
       * classe abstraite. C# génère une erreur à la compilation si vous
       * ne mettez pas en commentaire la ligne qui suit
       */
      // AbstractBaseClass ba = new AbstractBaseClass();
      // répète la même expérience avec Subclass1
      Console.WriteLine("Création d'un objet Subclass1");
      SubClass1 sc1 = new SubClass1();
      Test(sc1);
      // et enfin un objet Subclass2
      Console.WriteLine("\nCréation d'un objet Subclass2");
      SubClass2 sc2 = new SubClass2();
      Test(sc2);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

arabe, hindi, ni de toute autre langue. Le format de fichier Unicode, plus souple, bénéficie d'une compatibilité ascendante avec des caractères ANSI, et offre un assez grand nombre d'autres alphabets. Unicode existe en plusieurs formats, mais UTF8 en est le format par défaut pour C#.

Le programme `FileWrite` suivant lit des lignes de données sur la console et les écrits sur un fichier choisi par l'utilisateur :




```
// FileWrite - écrit dans un fichier texte
//           ce qui est saisi sur la console
using System;
using System.IO;
namespace FileWrite
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // crée l'objet de nom de fichier - la boucle while nous permet
      // de continuer à essayer avec différents noms de fichiers
      // jusqu'à ce que nous réussissions
      StreamWriter sw = null;
      string sFileName = "";
      while(true)
      {
        try
        {
          // saisie du nom du fichier de sortie (Entrée pour quitter)
          Console.Write("Entrez un nom de fichier "
                      + "(Entrez un nom vide pour quitter):");
          sFileName = Console.ReadLine();
          if (sFileName.Length == 0)
          {
            // pas de nom de fichier - fait passer au-delà de la boucle
            // while par sécurité
            break;
          }
          // ouvre le fichier pour y écrire ; envoie une exception si
          // le fichier existe déjà :
          //   FileMode.CreateNew pour créer un fichier si
          //                   il n'existe pas déjà, ou envoie
          //                   une exception si le fichier existe
          //   FileMode.Append pour créer un nouveau fichier ou ajouter
          //                   quelque chose à un fichier existant
          //   FileMode.Create pour créer un nouveau fichier ou
          //                   pour tronquer un fichier existant
          //   les possibilités de FileAccess sont :
          //                   FileAccess.Read,
```

```
      Console.WriteLine("Valeur donnée par OutputFunction = {0}",
                        id.ToString());
    }
    // ToString - fournit une simple fonction de type string
    override public string ToString()
    {
      return "Class1 du programme StructureExample";
    }
  }
}
```

Ce programme met à l'épreuve la structure Int32.

Main() commence par créer un objet i de type int. Main() utilise le constructeur par défaut Int32() (mais vous pourriez dire le constructeur int()) pour initialiser i à 0. Le programme poursuit en assignant une valeur à i. Il est évident que cela diffère légèrement du format que vous utiliseriez pour créer vous-même une structure.

Main() passe la variable i à OutputFunction(), qui est déclarée pour accepter un objet implémentant l'interface IFormattable. Celle-ci est la même que l'interface IDisplayable que j'ai définie dans d'autres programmes (la seule méthode de IFormattable est ToString). Toutes les classes et toutes les structures héritent par Object de l'interface IFormattable.

OutputFunction() dit à l'objet IFormattable de s'afficher lui-même (la variable Int32 n'a aucun problème parce qu'elle a sa propre méthode ToString()). Cela est démontré encore plus clairement dans l'appel à OutputFunction(2). Étant de type Int32, la constante 2 implémente également IFormattable. Enfin, rien que pour vous le mettre sous les yeux, Main() invoque directement 3.ToString(). La sortie de cette première section de Main() est :

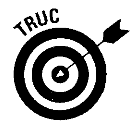
```
Valeur donnée par OutputFunction = 1
Valeur donnée par OutputFunction = 2
Extrait directement = 3
```

Le programme entre maintenant dans une section sans équivalent. Main() déclare un tableau d'objets de type Object. Il stocke un objet string dans le premier élément, un objet int dans le deuxième, une instance de Class1 dans le troisième, et ainsi de suite. Cela est autorisé, parce que String, Int32, et Class1 dérivent tous d'Object.

Le programme fait alors une boucle sur les objets du tableau. Main() est capable d'aller chercher les entiers en demandant à chaque objet s'il

Les classes d'I/O sont décrites dans l'espace de nom `System.IO`. La classe de base des I/O de fichier est `FileStream`. Autrefois, le programmeur ouvrait un fichier. La commande `open` préparait le fichier et retournait un handle. En général, ce handle n'était rien de plus qu'un numéro d'identification. Chaque fois qu'on voulait faire une opération de lecture ou d'écriture sur ce fichier, il fallait présenter ce numéro.

C# utilise une approche plus intuitive. Il associe chaque fichier à un objet de la classe `FileStream`. Le constructeur de `FileStream` ouvre le fichier. Les méthodes de `FileStream` effectuent les opérations d'I/O sur le fichier.



`FileStream` n'est pas la seule classe qui peut effectuer des opérations d'I/O sur des fichiers, mais c'est bien elle qui représente notre bon vieux fichier de base, qui correspond à 90 % de nos besoins d'I/O sur les fichiers. C'est la classe racine qui est décrite dans cette section. Si elle est satisfaisante pour C#, elle l'est aussi pour moi.

`FileStream` est une classe très basique. Ouvrir un fichier, fermer un fichier, lire un bloc et écrire un bloc, c'est à peu près tout ce qu'elle vous donne. Heureusement, l'espace de nom `System.IO` contient un ensemble de classes qui complètent `FileStream` pour vous donner un accès plus facile aux fichiers, et un sentiment de confort douillet :

- `ReadBinary/WriteBinary` : Ce sont deux classes de flux qui contiennent des méthodes permettant de lire et d'écrire tous les types valeur : `ReadChar()`, `WriteChar()`, `ReadByte()`, `WriteByte()`, et ainsi de suite. C'est utile pour écrire un objet en format binaire (non lisible par un être humain).

- `TextReader/TextWriter` : Deux classes permettant de lire et d'écrire des caractères (du texte). Elles se présentent en deux versions : `StringReader/StringWriter` et `StreamReader/StreamWriter`.

- `StringReader/StringWriter` : Une simple classe de flux qui se contente de lire et d'écrire des chaînes.

- `StreamReader/StreamWriter` : Une classe plus sophistiquée de lecture et d'écriture de texte pour ceux qui en veulent plus.

Cette section fournit les programmes suivants, qui montrent comment utiliser ces classes : `FileWrite` et `FileRead`.

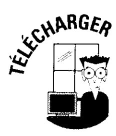
exemple, int n'est que l'autre nom de la structure Int32, double est l'autre nom de la structure Double, et ainsi de suite. Le Tableau 14.1 donne la liste complète des types et leur nom de structure correspondant.

Tableau 14.1 : Les noms de structure des types de variable intrinsèques.

Nom de type	Nom de structure
bool	Boolean
byte	Byte
sbyte	SByte
char	Char
decimal	Decimal
double	Double
float	Single
int	Int32
uint	UInt32
long	Int64
ulong	UInt64
object	Object
short	Int16
ushort	UInt16

Comment le système de types est-il unifié par des structures communes ? Un exemple

int n'est que l'autre nom de la structure Int32. Comme toutes les structures dérivent d'Object, int doit en dériver comme les autres. Cela conduit à quelques résultats fascinants, comme le montre le programme suivant :




```
// TypeUnification - montre comment int et Int32
//                   sont en fait la même chose
using System;
namespace TypeUnification
{
```

```
namespace AccessControlLib
{
  using System;
  public class Class2
  {
    public void A_public()
    {
      Console.WriteLine("Class2.A_public");
    }
    protected void B_protected()
    {
      Console.WriteLine("Class2.B_protected");
    }
    private void C_private()
    {
       Console.WriteLine("Class2.C_private");
    }
    internal void D_internal()
    {
      Console.WriteLine("Class2.D_internal");
    }
    internal protected void E_internalprotected()
    {
      Console.WriteLine("Class2.E_internalprotected");
   }
  }
}
```

Le programme `AccessControl` est constitué de `Class1` et `Class3`, qui sont contenues dans l'espace de nom `AccessControl`, et de la classe `Class2` de l'espace de nom `AccessControlLib`. Les appels aux méthodes dans `Class1.Main()` mettent en évidence chaque type d'accès :

- Les méthodes déclarées comme `public` sont accessibles à toutes les méthodes de tous les espaces de nom. Aussi, `Class1` peut invoquer directement `Class2.A_public()`.

- Les méthodes déclarées comme `protected` sont accessibles depuis la classe `Class2` et toute classe qui hérite de `Class1`. L'appel `class1.B_protected()` est autorisé, parce que `Class1` hérite de `Class2`. L'appel `class3.B_protected()` est illicite.

- Les méthodes déclarées `private` ne sont accessibles qu'aux autres membres de la même classe. Aussi, l'appel `class2.C_private()` n'est pas autorisé, alors que l'appel `class1.C_private()` est correct.

```
    // ChangeReferenceFunction - passe la struct par référence
    public static void ChangeReferenceFunction(ref Test t,
                            int newValue, double dNewValue)
    {
      t.N = newValue;
      Test.D = dNewValue;
    }
    // OutputFunction - affiche toute méthode qui implémente
    //                  ToString()
    public static void OutputFunction(IDisplayable id)
    {
      Console.WriteLine("id = {0}", id.ToString());
    }
  }
}
```

Le programme `StructureExample` définit tout d'abord une interface, `IDisplayable`, puis une simple structure, `Test`, qui implémente cette interface. `Test` définit également deux membres : un membre instance, `n`, et un membre statique, `d`. Un initialiseur statique assigne la valeur 20 à `d`. Toutefois, le membre instance `n` n'a pas droit à un initialiseur.

La structure `Test` définit un constructeur, une propriété instance `N`, et une propriété statique `D`.

`Test` définit sa propre méthode, `ChangeMethod()`, ainsi que la redéfinition de la méthode `ToString()`. En fournissant `ToString()`, `Test` implémente l'interface `IDisplayable`.

La fonction `Main()` met `Test` à l'épreuve. Tout d'abord, elle crée un objet `test` dans la mémoire locale, et utilise le constructeur pour initialiser cet espace. `Main()` appelle alors `OutputFunction()` pour afficher l'objet.

`Main()` appelle ensuite la fonction `ChangeValueFunction()`, lui passant `test` avec deux constantes numériques. `ChangeValueFunction()` assigne ces deux valeurs aux membres `n` et `d` de `Test`. Une fois que cette fonction a retourné ses résultats, `OutputFunction()` révèle que `d` a été modifié, alors que `n` ne l'a pas été.

L'appel à `ChangeValueFunction()` passe par valeur l'objet `test` de type structure. À l'intérieur de cette fonction, l'objet `t` est une copie du `test` original, et non l'objet lui-même. Aussi, l'assignation à `t.N` change la copie locale, mais n'a aucun effet sur `test` de retour dans `Main()`. Toutefois, tous les objets de la classe `Test` partagent le même membre statique `d`. Aussi, l'assignation à `Test.D` change `d` pour tous les objets, y compris `test`.

Contrôler l'accès aux classes avec les espaces de nom

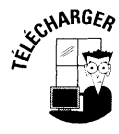
Les espaces de nom autorisent un certain niveau d'indépendance dans des ensembles de classes qui n'ont pas grand-chose à voir entre elles. Par exemple, si vous travaillez sur un ensemble de classes mathématiques, vous pouvez utiliser une classe comme conteneur pour y stocker des ensembles de valeurs.



Le niveau d'indépendance est appelé le *niveau de couplage*. Une classe qui accède aux membres internes d'une autre classe est dite *fortement couplée*. Des classes qui n'accèdent l'une à l'autre que par des méthodes publiques sont dites *faiblement couplées*.

Le Chapitre 11 montre comment des descripteurs `public`, `protected` et `private` séparent les classes dans un même espace de nom. L'ajout du mot-clé `internal` spécifie qu'un objet est accessible à partir du même espace de nom mais pas aux classes externes. Les membres spécifiés comme `internal protected` sont accessibles à la fois aux classes du même espace de nom et aux sous-classes.

Le programme `AccessControl` suivant montre le fonctionnement de l'ensemble complet des contrôles d'accès :



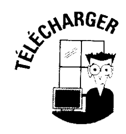

```
// AccessControl - montre les différentes formes
//                 de contrôle d'accès
namespace AccessControl
{
  using System;
  using AccessControlLib;
  public class Class1 : Class2
  {
    public static int Main(string[] strings)
    {
      Class1 class1 = new Class1();
      Class2 class2 = new Class2();
      Class3 class3 = new Class3();
      // les méthodes publiques sont accessibles par d'autres classes
      // dans d'autres espaces de nom
      class2.A_public();
      // les méthodes protégées sont accessibles à travers
      // la hiérarchie d'héritage
      class1.B_protected();
      //class3.B_protected();
      // les méthodes privées ne sont accessibles que par
```



Toutes les classes héritent d'`Object` qu'elles le disent explicitement ou non. Vous pouvez redéfinir les méthodes d'`Object`. En termes pratiques, la seule méthode que vous pouvez vouloir redéfinir est `ToString()`. Celle-ci permet à l'objet de créer une représentation affichable de lui-même.

Mettre une structure à l'épreuve par l'exemple

L'exemple de programme suivant montre les différentes caractéristiques d'une structure :




```
// StructureExample - montre les différentes propriétés
//                    d'un objet struct
using System;
using System.Collections;
namespace StructureExample
{
  public interface IDisplayable
  {
    string ToString();
  }
  public struct Test : IDisplayable
  {
    // une struct peut avoir des membres donnée
    // objet et de classe (statiques)
    // les membres statiques peuvent avoir des initialiseurs
    private int n;
    private static double d = 20.0;
    // on peut utiliser un constructeur pour initialiser
    // les membres donnée d'une struct
    public Test(int n)
    {
      this.n = n;
    }
    // une struct peut avoir des propriétés d'objet
    // et des propriétés de classe (statiques)
    public int N
    {
      get { return n;}
      set { n = value; }
    }
    public static double D
    {
      get { return d; }
      set { d = value; }
    }
```

```
    }
  }
  namespace Paint
  {
    public class PaintColor
    {
      public PaintColor(int nRed, int nGreen, int nBlue) {}
      public void Paint() {}
      public static void StaticPaint() {}
    }
  }
  namespace MathRoutines
  {
    public class Test
    {
      static public void Main(string[] args)
      {
        // crée un objet de type Sort dans l'espace de nom
        // où nous nous trouvons et invoque une fonction
        Sort obj = new Sort();
        obj.SomeFunction();
        // crée un objet dans un autre espace de nom, remarquez que
        // le nom de l'espace de nom doit figurer explicitement dans toute
        // référence de classe
        Paint.PaintColor black = new Paint.PaintColor(0, 0, 0);
        black.Paint();
        Paint.PaintColor.StaticPaint();
      }
    }
  }
```

Dans ce cas, les deux class `Sort` et `Test` sont contenues dans le même espace de nom, `MathRoutines`, bien qu'elles apparaissent dans des déclarations différentes dans le module.



Normalement, `Sort` et `Test` seraient dans des modules source C# différents que vous pourriez générer ensemble pour en faire un seul programme.

La fonction `Test.Main()` peut référencer la classe `Sort` sans spécifier l'espace de nom parce que les deux classes sont dans le même espace de nom. Toutefois, `Main()` doit spécifier l'espace de nom `Paint` quand elle se réfère à `PaintColor`, comme dans l'appel à `Paint.PaintColor.StaticPaint()`. Remarquez que vous n'avez pas besoin de prendre des précautions spéciales en vous référant à `black.Paint()`, parce que la classe et l'espace de nom de l'objet `black` sont connus.

de référence nécessite que le programme invoque 101 fois new (une fois pour le tableau et une fois pour chaque objet) :

```
MyClass[] mc = new MyClass[100];
for(int i = 0; i < ms.Length; i++)
{
  mc[i] = new MyClass();
}
mc[0].n = 0;
```

Ce tableau est également un gros consommateur de ressources, de temps comme d'espace. Tout d'abord, chaque élément du tableau mc doit être assez vaste pour contenir une référence à un objet. En outre, chaque objet MyClass fait une consommation invisible de ressources au-dessus et au-delà du seul membre donnée n. Enfin, il y a le temps que prend le programme pour effectuer toutes les manœuvres nécessaires afin de réduire cent fois de suite un petit bloc de mémoire.

La mémoire destinée aux objets de type structure est allouée en tant que partie du tableau :

```
// déclaration d'un tableau du simple type valeur int
int[] integers = new int[100];
integers[0] = 0;
// la déclaration d'un tableau de struct est tout aussi simple
MyStruct[] ms = new MyStruct[100];
ms[0].n = 0;
```

Le constructeur de structure

Une structure peut être initialisée en utilisant une syntaxe semblable à celle des classes, ce qui est intéressant :

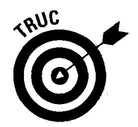
```
public struct MyStruct
{
  public int n;
  public double d;
}
MyStruct ms = new MyStruct();
```

En dépit des apparences, cela n'alloue pas un bloc de mémoire à partir du tas, mais initialise seulement n et d à la valeur zéro.



Un *fichier de projet* contient des instructions sur les fichiers qui constituent le projet et la manière dont ils se combinent.

Vous pouvez combiner des fichiers de projet pour produire des combinaisons de programme qui dépendent des mêmes classes définies par l'utilisateur. Par exemple, vous pouvez vouloir associer un programme d'écriture avec le programme de lecture correspondant. De cette manière, si l'un des deux change, l'autre est automatiquement régénéré. L'un des projets décrirait le programme d'écriture, et l'autre décrirait le programme de lecture. On appelle *solution* un ensemble de fichiers de projet.



Un programmeur Visual C# utilise l'Explorateur de Visual Studio pour combiner en projets des fichiers source C# dans l'environnement Visual Studio.

Réunir des fichiers source dans un espace de nom

Vous avez la possibilité de réunir des classes communes dans un espace auquel a été assigné un nom significatif. Par exemple, vous pouvez compiler toutes les routines dont la signification est mathématique dans un espace de nom `MathRoutines`.



Il est possible, mais très improbable, de diviser un même fichier en plusieurs espaces de noms. Il est plus courant de regrouper plusieurs fichiers dans un même espace de nom. Par exemple, le fichier `Point.cs` peut contenir la classe `Point` et la classe `ThreeDSpace.cs` pour décrire les propriétés d'un espace euclidien. Vous pouvez combiner `Point.cs`, `ThreeDSpace.cs`, et d'autres fichiers source dans l'espace de nom `MathRoutines`.

Un espace de nom sert à plusieurs choses. C'est d'abord une réunion de classes. En tant que programmeur, vous pouvez raisonnablement supposer que les classes qui constituent l'espace de nom `MathRoutines` ont toutes quelque chose à voir avec des fonctions mathématiques. Par voie de conséquences, si vous recherchez une certaine fonction mathématique, c'est dans les classes qui constituent `MathRoutines` que vous pouvez aller la chercher en premier.

Un espace de nom évite les possibilités de conflit de nom. Par exemple, une bibliothèque d'entrées/sorties pour fichiers peut contenir une classe `Convert` qui convertit une représentation d'un type de fichier en un autre type. En même temps, une bibliothèque de traduction peut contenir une classe du même nom. L'assignation des espaces de nom `FileIO` et

Mais une référence à un objet est déclarée et initialisée d'une manière complètement différente :

```
public class MyClass
{
  public int n;
}
MyClass mc;
mc = new MyClass();
```



La variable de classe mc est appelée un *type référence*, parce qu'elle se réfère à une zone de mémoire potentiellement distante. Une variable intrinsèque de type int ou double est appelée *variable de type valeur.*

Toutefois, si vous examinez plus attentivement n et mc, vous verrez que la seule véritable différence est que C# alloue automatiquement la mémoire pour une variable de type valeur, alors que vous devez allouer la mémoire pour un objet de classe. N'y a-t-il rien qui puisse réunir les deux dans une Théorie unifiée des classes ?

La structure C#

C# définit un troisième type de variable, appelé une *structure*, qui comble le fossé entre les types référence et les types valeur.

La syntaxe d'une déclaration de structure ressemble à celle d'une classe :

```
public struct MyStruct
{
  public int n;
  public double d;
}
public class MyClass
{
  public int n;
  public double d;
}
```

On accède à un objet structure comme à un objet de classe, mais l'allocation est identique à celle d'un type valeur :

```
// déclare et accède à une variable d'un type valeur simple
int n;
```

Chapitre 16

Manipuler des fichiers en C#

Dans ce chapitre :

Gérer plusieurs fichiers source pour un même programme.

Lire et écrire des fichiers de données.

En C#, l'*accès aux fichiers* signifie deux choses différentes. La plus évidente est l'enregistrement et la récupération de données sur le disque. Il y a toutefois une autre signification qui concerne la manière dont le code source C# est regroupé dans des fichiers source.

Les fonctions permettent de diviser une longue chaîne de code en unités séparables et maintenables. L'organisation en classes permet de regrouper les données comme les fonctions de façon pertinente afin de réduire encore la complexité du programme. C# offre un autre niveau de regroupement : il vous permet de regrouper des classes similaires dans une bibliothèque séparée.

Diviser un même programme en plusieurs fichiers source

Les programmes de ce livre ne sont faits que dans un but de démonstration. Ils ne dépassent pas quelques dizaines de lignes de long et ne contiennent qu'un petit nombre de classes. Un programme de niveau industriel, avec tous les aspects décoratifs nécessaires, peut comporter des centaines de milliers de lignes de code, réparties dans plusieurs centaines de classes.

Il devient rapidement impraticable de stocker toutes ces classes dans un même module. Pour commencer, il y a l'exigence de maintenir les classes bien en ordre. Ensuite, le travail de réalisation de grands programmes est généralement réparti entre de nombreux programmeurs. Le même fichier

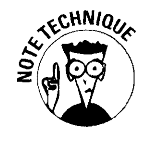
```
    }
  }
  public class Class1
  {
    public static int Main(string[] strings)
    {
      SubClass sc1 = new SubClass(10);
      SubClass sc2 = new SubClass(20);
      MyFunc(sc1, sc2);
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
    // MyFunc - utilise les méthodes fournies par l'interface ICompare
    //          pour afficher la valeur de deux objets, puis
    //          l'indication de celui qui est le plus grand (selon
    //          l'objet lui-même)
    public static void MyFunc(ICompare ic1, ICompare ic2)
    {
      Console.WriteLine("La valeur de ic1 est {0} et celle de ic2 est {1}",
                        ic1.GetValue(), ic2.GetValue());
      string s;
      switch (ic1.CompareTo(ic2))
      {
        case 0:
          s = "est égal à";
          break;
        case -1:
          s = "est plus petit que";
          break;
        case 1:
          s = "est plus grand que";
          break;
        default:
          s = "quelque chose qui cloche";
          break;
      }
      Console.WriteLine(
             "Les objets eux-mêmes considèrent que ic1 {0} ic2", s);
    }
  }
}
```

`AbstractInterface` est encore un programme un peu long, mais relativement simple.

L'interface `ICompare` décrit une classe qui peut comparer deux objets et aller chercher leur valeur. `ICompare` hérite de l'interface `IComparable`

Le remplacement de ces fonctions signifie que même les fonctions conçues pour attraper les exceptions de la classe générique `Exception` n'ont qu'un accès limité aux nouveaux membres donnée.

En commençant par `Main()`, le programme crée un objet `MathClass` dont la valeur est 0, puis essaie astucieusement d'en prendre l'inverse. Je ne sais pas si vous avez déjà essayé, mais je n'ai jamais vu beaucoup d'inverses de 0, et si ma fonction était censée retourner un nombre, ça me rendrait un peu méfiant.

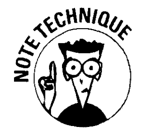


Le processeur Intel retourne effectivement une valeur pour 1.0/0.0 : `Infinity`. Il existe plusieurs valeurs spéciales en virgule flottante pour traiter de tels cas plutôt que d'envoyer une exception, car tous les langages n'ont pas la capacité de traiter des exceptions. Parmi ces valeurs spéciales, il y a l'infini positif et négatif, et le symbole `NaN` (Not_a_Number – pas un nombre) positif et négatif.

Dans les circonstances normales, la méthode `Inverse()` retourne le résultat attendu. Quand on lui passe une valeur nulle, cette méthode aux idées larges envoie une `CustomException`, passant une chaîne d'explications avec l'objet fautif.

Travaillant à l'envers, `Main()` attrape l'exception, puis affiche un bref message destiné à expliquer où en est le message dans son traitement : "`Erreur fatale inconnue`" signifie probablement que le programme est sur le point de fermer boutique et de rentrer chez lui. `Main()` donne ensuite à l'exception la possibilité de s'expliquer en invoquant sa méthode `ToString()`. (Voyez l'encadré "ToString(), la carte de visite de la classe" dans ce chapitre.)

Comme dans ce cas l'objet exception est effectivement un `CustomException`, le contrôle passe à `CustomException.ToString()`. Cette méthode `ToString()` affiche le message de l'exception avec la méthode cible initiale et le numéro de ligne correspondant.



`Message()` est une méthode virtuelle d'`Exception`, dont toute classe d'exceptions personnalisée doit hériter.

Plutôt que de faire des hypothèses hasardeuses, `Message()` permet également à l'objet `MathClass` de s'afficher lui-même en utilisant sa méthode `ToString()`. `MathClass.ToString()` retourne une chaîne contenant la valeur et la description de l'objet.



Ne supposez rien de plus que ce que vous savez. C'est sur la méthode `ToString()` d'un objet que vous devez compter pour créer une version `string` de celui-ci, plutôt que d'essayer d'accéder à l'objet lui-même pour en extraire les valeurs à afficher.

```
Tri de la liste des oiseaux
Alouette
Corbeau
Étourneau
Grive
Hirondelle
Tourterelle
Vautour
Appuyez sur Entrée pour terminer...
```

Les étudiants et les oiseaux sont triés, dans la logique de leurs catégories respectives.

Héritage et interface

Une interface peut "hériter" des méthodes d'une autre interface. Je mets des guillemets autour du mot *hériter*, parce qu'il ne s'agit pas d'un véritable héritage, même s'il en a pourtant l'air :

```
// ICompare - interface capable de se comparer elle-même
//            et d'afficher sa propre valeur
public interface ICompare : IComparable
{
  // GetValue - retourne sa propre valeur sous forme d'un int
  int GetValue();
}
```

L'interface `ICompare` hérite de `IComparable` l'exigence d'implémenter la méthode `CompareTo()`. À cela, elle ajoute l'exigence d'implémenter `GetValue()`. Un objet `ICompare` peut être utilisé comme un objet `IComparable`, car, par définition, le premier implémente les exigences du second. Toutefois, il ne s'agit pas là d'un héritage complet au sens C#, orienté objet, de ce terme. Le polymorphisme n'est pas possible. De plus, les relations de constructeurs ne s'appliquent pas.

Je donne une démonstration de l'héritage d'interface dans le programme `AbstractInterface`, dans la section suivante.

Rencontrer une interface abstraite

Afin d'implémenter une interface, une classe doit redéfinir chaque méthode de celle-ci. Toutefois, une classe peut redéfinir une méthode d'une interface par une méthode abstraite (naturellement, bien sûr, une telle classe est abstraite) :

```
using System;
namespace CustomException
{
  public class CustomException : Exception
  {
    private MathClass mathobject;
    private string sMessage;
    public CustomException(string sMsg, MathClass mo)
    {
      mathobject = mo;
      sMessage =  sMsg;
    }
    override public string Message
    {
      get{return String.Format("Le message est <{0}>, l'objet est {1}",
                               sMessage, mathobject.ToString());}
    }
    override public string ToString()
    {
      string s = Message;
      s += "\nException envoyée par";
      s += TargetSite;
      return s;
    }
  }
  // MathClass - collection de fonctions mathématiques
  //             de ma création (pas encore grand-chose à montrer)
  public class MathClass
  {
    private int nValueOfObject;
    private string sObjectDescription;
    public MathClass(string sDescription, int nValue)
    {
      nValueOfObject = nValue;
      sObjectDescription = sDescription;
    }
    public int Value {get {return nValueOfObject;}}
    // Message - affiche le message avec la valeur de
    //           l'objet MathClass attaché
    public string Message
    {
      get
      {
        return String.Format("({0} = {1})",
          sObjectDescription,
          nValueOfObject);
      }
    }
    // ToString - combine le message personnalisé avec
```

```
    }
    // implémentent l'interface IDisplayable :
    // GetString - retourne le nom de l'oiseau
    public string GetString()
    {
      return Name;
    }
  }
}
```

La class `Student` (elle est à peu près au milieu de ce listing) implémente les interfaces `IComparable` et `IDisplayable` que nous avons décrites plus haut. `CompareTo()` compare les étudiants par "grade", ce qui a pour conséquence que les étudiants sont triés par grade. `GetString()` retourne le nom et le grade de l'étudiant.

Parmi les autres méthodes de `Student`, il y a les propriétés en lecture seule `Name` et `Grade`, un constructeur simple, et une méthode `CreateStudentList()`. Cette dernière retourne simplement une liste fixe d'étudiants (au départ, j'avais pensé permettre à l'utilisateur d'entrer au clavier les noms des étudiants, mais le listing devenait si gros qu'on n'y voyait plus l'essentiel).

La classe `Bird`, en bas du listing, implémente également les interfaces `IComparable` et `IDisplayable`. Elle implémente `CompareTo()` en comparant des noms d'oiseaux à l'aide d'une méthode similaire intégrée à la classe `String`. Ainsi, un oiseau est "plus grand" qu'un autre si son nom est plus grand. Cette méthode permet de trier les oiseaux par ordre alphabétique. La méthode `GetName()` retourne simplement le nom de l'oiseau.

Nous sommes maintenant prêts à revenir dans `Main()`, au bon endroit. La méthode `CreateStudentList()` est utilisée pour retourner une liste non triée, qui est stockée dans le tableau `students`.



Pour nommer une collection d'objets, comme un tableau, utilisez un nom au pluriel.

Ce tableau d'étudiants est d'abord introduit dans un tableau `comparableObjects`. Celui-ci est différent des tableaux utilisés dans les autres chapitres (en particulier ceux du Chapitre 6). Ceux-ci sont des tableaux d'objets d'une classe particulière, par exemple un tableau d'objets `Student`, alors que `comparableObjects` est un tableau d'objets qui implémentent l'interface `IComparable`, indépendamment de la classe à laquelle ils appartiennent.

Le tableau `comparableObjects` est passé à la méthode `Array.Sort()`, qui en trie les éléments sur la base du grade.

```
public void f1()
{
  try
  {
    f2();
  }
  // attrape une erreur . . .
  catch(MyException me)
  {
    // ... traite une partie de l'erreur . . .
    Console.WriteLine("Exception MyException attrapée dans f1()");
    // . . . génère maintenant une nouvelle exception
    //  pour remonter la chaîne de transmission
    throw new Exception("Erreur envoyée parf1()");
  }
}
```

Envoyer un nouvel objet exception permet à une classe de formuler un nouveau message d'erreur avec des informations supplémentaires, tout en clarifiant ce qu'il pouvait y avoir d'approximatif au départ. Passer un objet `Exception` générique à la place d'un objet spécialisé `MyException` garantit que l'exception sera traitée à un niveau situé au-dessus de `f1()`.

Envoyer une nouvelle exception présente l'inconvénient que l'indication de pile redémarre au point du nouvel envoi. La source de l'erreur originale est donc perdue, à moins que `f1()` n'ait pris des précautions spéciales pour la conserver.

Une commande `throw` seule, sans argument, renvoie le même objet exception :

```
public void f1()
{
  try
  {
    f2();
  }
  // attrape une erreur . . .
  catch(Exception e)
  {
    // ... traite une partie de l'erreur . . .
    Console.WriteLine("Exception attrapée dansf1()");
    // . . . poursuite du cheminement de l'exception
    throw;
  }
}
```

```
public string Name
{
  get
  {
    return sName;
  }
}
public double Grade
{
  get
  {
    return dGrade;
  }
}
// implémente l'interface IComparable :
// CompareTo - compare à un autre objet (dans ce cas,
//             des objets Student) et décide lequel
//             vient après l'autre dans le
//             tableau trié
public int CompareTo(object rightObject)
{
  // compare l'objet Student courant (appelons-le
  // 'celui de gauche') à l'autre student (appelons-le
  // 'celui de droite'), et génère une erreur si aucun des deux
  // n'est un objet Student
  Student leftStudent = this;
  if (!(rightObject is Student))
  {
    Console.WriteLine
      ("Méthode de comparaison à laquelle est passé un nonStudent");
    return 0;
  }
  Student rightStudent = (Student)rightObject;
  // génère maintenant -1, 0, ou 1 sur la base du
  // critère de tri (la moyenne des points d'UV de l'étudiant)
  // (la classe Double contient une méthode CompareTo()
  // que nous aurions pu utiliser à la place)
  if (rightStudent.Grade < leftStudent.Grade)
  {
    return -1;
  }
  if (rightStudent.Grade > leftStudent.Grade)
  {
    return 1;
  }
  return 0;
}
// implémentent l'interface IDisplayable :
// GetString - retourne une représentation de l'étudiant
```

```
            Console.Read();
        }
    }
}
```

Main() crée un objet Class1 et l'utilise immédiatement pour invoquer la méthode f1(). Cette méthode appelle f2(), qui appelle f3(), qui appelle f4(). La fonction f4() effectue des vérifications d'erreur extrêmement sophistiquées, qui la conduisent à envoyer soit un objet MyException, soit un objet Exception générique, selon la valeur de l'argument booléen. L'exception est d'abord passée à f3(). Là, C# ne trouve aucune instruction catch, et le contrôle remonte donc à f2(), qui attrape l'objet MyException. Comme l'objet Exception générique n'a pas encore trouvé d'instruction catch correspondante, le contrôle continue à remonter. Finalement, c'est f1() qui contient une instruction catch correspondant à l'objet envoyé.

Le deuxième appel de Main() provoque l'envoi par f4() d'un objet MyException, qui est attrapé par f2(). Cette exception n'est pas envoyée à f1(), parce qu'elle a été attrapée et traitée par f2().

Le programme donne la sortie suivante :

```
Envoie d'abord une exception générique
Exception générique attrapée dans f1()
Exception générique envoyée dans f4()

Envoie d'abord une exception spécifique
Exception MyException attrapée dans f2()
MyException envoyée dans f4()
Appuyez sur Entrée pour terminer...
```

Une fonction comme f3(), qui ne contient aucune instruction catch, n'a rien d'inhabituel. Je pourrais même dire que la plupart des fonctions ne contiennent aucune instruction catch, mais je pourrais tout aussi bien ne pas le dire. Une fonction n'a aucune raison d'attraper une exception si elle ne contient rien qui lui permette de traiter l'erreur d'une manière pertinente. Imaginez une fonction mathématique ComputeX() qui appelle Factorial() pour effectuer certains de ses calculs. En supposant que son code interne soit correct, si Factorial() envoie une exception, c'est parce que la fonction appelante lui a passé une valeur incorrecte. ComputeX() peut ou non être capable d'identifier pourquoi cette valeur était incorrecte, mais en tous les cas, elle ne peut certainement pas résoudre le problème.

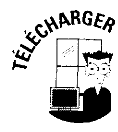
Une fonction comme f2() n'attrape qu'un seul type d'exception. Elle recherche une certaine classe d'erreurs. Par exemple, MyException peut faire partie des



Comparez cet exemple à l'algorithme de tri du Chapitre 6. L'implémentation de cet algorithme demandait pas mal de travail et nécessitait beaucoup de code. Il est vrai que rien ne vous garantit que `Array.Sort()` est meilleur ou plus rapide que cet algorithme. Il est seulement plus facile.

Assembler le tout

Voilà le moment que vous attendiez tous : le programme `SortInterface` complet, construit en utilisant ce que nous avons décrit plus haut dans ce chapitre :

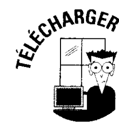



```
// SortInterface - montre comment
//                 on peut utiliser le principe de l'interface pour offrir
//                 une meilleure souplesse pour le factoring
//                 et l'implémentation des classes
using System;
namespace SortInterface
{
  // IDisplayable - objet capable de se convertir lui-même en
  //                une chaîne affichable
  interface IDisplayable
  {
    // GetString - retourne votre représentation sous forme de chaîne
    string GetString();
  }
  class Class1
  {
    public static void Main(string[] args)
    {
      // trie les étudiants par leur moyenne de points d'UV...
      Console.WriteLine("Tri de la liste des étudiants");
      // reçoit un tableau d'étudiants non trié
      Student[] students = Student.CreateStudentList();
      // utilise l'interface IComparable pour trier
      // le tableau
      IComparable[] comparableObjects = (IComparable[])students;
      Array.Sort(comparableObjects);
      // l'interface IDisplayable affiche maintenant les résultats
      IDisplayable[] displayableObjects = (IDisplayable[])students;
      DisplayArray(displayableObjects);
      // trie maintenant par nom un tableau d'oiseaux en utilisant
      // les mêmes routines, bien que les classes Bird
      // et Student n'aient pas de classe de base commune
      Console.WriteLine("\nTri de la liste des oiseaux");
      Bird[] birds = Bird.CreateBirdList();
      // remarquez qu'il n'est pas indispensable de faire un cast
```

Laisser quelques envois vous filer entre les doigts

Et si C# part à la recherche d'une instruction `catch` dans la fonction appelante correspondant à l'objet exception envoyé, et n'en trouve aucune qui corresponde ? Et si la fonction appelante ne contient aucune instruction `catch` ? Que faire ?

Examinez cette simple succession d'appels de fonctions :



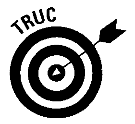

```
// MyException - montre comment une nouvelle classe d'exceptions peut
//               être créée, et comment des fonctions peuvent
//               attraper exactement ce qu'elles sont faites
//               pour traiter, tout en laissant passer les autres
using System;
namespace MyException
{
  // introduit un certain type de 'MyClass'
  public class MyClass{}
  // MyException -  - ajoute à la classe Exception standard
  //                  une référence à MyClass
  public class MyException : Exception
  {
    private MyClass myobject;

    public MyException(string sMsg, MyClass mo) : base(sMsg)
    {
      myobject = mo;
    }

    // permet aux classes extérieures d'accéder à une classe d'information
    public MyClass MyCustomObject{ get {return myobject;}}
  }

  public class Class1
  {
    // f1 -  - attrape tout objet Exception générique
    public void f1(bool bExceptionType)
    {
      try
      {
        f2(bExceptionType);
      }
      catch(Exception e)
      {
        Console.WriteLine("Exception générique attrapée dans f1()");
```

```
Bart      :50
Lisa      :100
Maggie    :30
```



Que c'est beau ! Voyez comme les résultats sont alignés sur les noms complétés par des espaces pour faire tous la même longueur.

Interfaces prédéfinies

De même, vous trouverez dans la bibliothèque standard de C# des interfaces intégrées en abondance. Par exemple, l'interface `IComparable` est définie de la façon suivante :

```
interface IComparable
{
  // compare l'objet courant à l'objet 'o' ; retourne
  // 1 s'il est plus grand, - s'il est plus petit, 0 dans les autres cas
  int CompareTo(object o);
}
```

Une classe implémente l'interface `IComparable` en implémentant une méthode `CompareTo()`. Par exemple, `String()` implémente cette méthode en comparant deux chaînes. Si les chaînes sont identiques, elle retourne 0. Si elles ne le sont pas, elle retourne soit 1 soit un signe -, selon la plus "grande" des deux chaînes.



Si vous voulez savoir comment une chaîne peut être "plus grande" qu'une autre, voyez le Chapitre 9.

N'y voyez aucune intention darwinienne, mais vous pouvez dire qu'un objet `Student` est "plus grand" qu'un autre objet `Student` si sa moyenne de points d'UV (grade point average) est plus grande. C'est soit un étudiant plus brillant, soit un meilleur courtisan, peu importe.

Implémenter la méthode `CompareTo()` implique que les objets peuvent être triés. Si un étudiant est "plus grand" qu'un autre, vous devez pouvoir trier les étudiants du "plus petit" au "plus grand". En fait, la classe `Array` implémente déjà la méthode suivante :

```
Array.Sort(IComparable[] objects);
```

De retour dans `Main()`, l'instruction `catch` spécifie qu'elle attend un objet `MyException`. Une fois l'exception attrapée, le code de l'application peut encore demander n'importe quelle propriété d'une `Exception`, comme dans l'appel à `ToString()`. Cette instruction `catch` peut également invoquer des méthodes de l'objet `MyClass` fautif stocké dans l'envoi (`throw`).

Assigner plusieurs blocs catch

Le fragment de code de la section précédente décrit le processus par lequel un objet `MyException` localement défini est envoyé et attrapé. Mais examinez à nouveau l'instruction `catch` utilisée dans cet exemple :

```
public void SomeFunction()
{
  try
  {
    SomeOtherFunction();
  }
  catch(MyException me)
  {
  }
}
```

Et si `SomeOtherFunction()` avait envoyé une simple `Exception` ou une exception d'un type autre que `MyException` ? Autant essayer d'attraper un ballon de football avec un filet à papillon. Le `catch` ne correspond pas à l'envoi. Heureusement, C# permet au programme de définir toutes sortes d'instructions `catch`, selon le type d'exception à attraper.

Les instructions `catch` doivent figurer l'une après l'autre après le bloc `try`, de la plus spécifique à la plus générale. C# teste chaque bloc `catch` en comparant séquentiellement les objets envoyés au type d'argument de l'instruction `catch`.

```
public void SomeFunction()
{
  try
  {
    SomeOtherFunction();
  }
  catch(MyException me)
  {
    // tous les objets MyException sont attrapés ici
```

Puis-je voir un programme qui PEUT_ÊTRE_UTILISÉ_COMME un exemple ?

Le programme SortInterface ci-dessous est une offre spéciale. Ses capacités, qui vous sont apportées par deux interfaces différentes, ne pourraient jamais être obtenues par une relation d'héritage. Une fois implémentées, les interfaces se tiennent prêtes à votre service.

Je veux toutefois diviser le programme SortInterface en sections, pour mettre en évidence différents principes. Je veux simplement faire en sorte que vous puissiez voir exactement comment fonctionne ce programme.

Créer votre interface "faites-le vous-même"

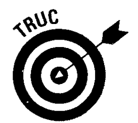
L'interface IDisplayable suivante sera satisfaite avec toute classe contenant une méthode GetString(). Cette méthode retourne une représentation sous forme de string de l'objet qui peut être affiché en utilisant WriteLine() :

```
// IDisplayable - objet qui implémente
//                la méthode GetString()
interface IDisplayable
{
  // retourne votre description
  string GetString();
}
```

La classe Student suivante implémente Idisplayable :

```
si class Student : IDisplayable
{
  private string sName;
  private double dGrade = 0.0;
  // accède aux méthodes en lecture seule
  public string Name
  {
    get
    {
      return sName;
    }
  }
  public double Grade
  {
```

Le reste de cette sortie est ce que l'on appelle une *indication de pile* (*stack trace*). La première ligne de l'indication de pile indique l'endroit à partir duquel l'exception a été envoyée. Dans ce cas : Factorial(int) (plus précisément, la ligne 23 du fichier source Class1.cs. Factorial(), ou la ligne 52 du même fichier, a été invoquée dans la fonction Main(string[]). L'indication de pile s'arrête à Main(), parce que c'est le module dans lequel l'exception a été attrapée.



L'indication de pile est disponible dans l'une des fenêtres du débogueur de Visual Studio. Je la décrirai au Chapitre 16.

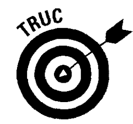
Il faut bien admettre que c'est plutôt impressionnant. Le message décrit le problème et identifie l'argument qui en est responsable. L'indication de pile vous dit à quel endroit l'exception a été envoyée, et comment le programme y est arrivé. Avec ces informations, vous pouvez vous jeter sur le problème comme la foudre.

Créer votre propre classe d'exceptions

La classe Exception standard fournie par la bibliothèque de classe de C# est capable de délivrer beaucoup d'informations. Vous pouvez demander à l'objet exception à quel endroit il a été envoyé, ainsi que toute chaîne demandée par la fonction qui signale l'erreur. Dans certains cas, toutefois, la classe Exception standard ne convient pas. Il peut y avoir trop d'informations pour qu'elles puissent tenir dans une seule chaîne. Par exemple, une fonction d'application peut vouloir se faire passer l'objet incriminé pour l'analyser.

Une classe localement définie peut hériter de la classe Exception, comme de n'importe quelle autre classe :

```
// CustomException - ajoute à la classe Exception standard
//                   une référence à MyClass
public class CustomException : Exception
{
  private MyClasss myobject;
  CustomException(string sMsg, MyClass mo) : base(sMsg)
  {
    myobject = mo;
  }
  // permet aux classes extérieures d'accéder à une classe d'information
  public MyClass MyCustomObject{ get {return myobject;}}
}
```



Par convention, faites commencer le nom d'une interface par la lettre *I*, et accolez-lui un adjectif. Comme d'habitude, ce ne sont là que les suggestions qui ont pour but de rendre vos programmes plus lisibles. C# vous permet d'utiliser les noms que vous voulez.

La déclaration suivante indique que la classe `PDA` implémente l'interface `Irecordable` :

```
public class PDA : IRecordable
{
  public void TakeANote(string sNote)
  {
    // . . . fait quelque chose pour enregistrer la note . . .
  }
}
```

Il n'y a pas de différence de syntaxe entre la déclaration d'une classe de base `ThingsThatRecord` et la déclaration qui implémente une interface `IRecordable`.



Voilà la principale raison d'être de la convention sur les noms d'interfaces : elle permet de reconnaître une interface d'une classe.

La conclusion est qu'une interface décrit une capacité. En tant que classe, j'obtiens mon brevet `IRecordable` lorsque j'implémente la capacité `TakeANote`.

Pourriez-vous me donner un exemple simple ?

Une classe implémente une interface en fournissant une définition pour chaque méthode de celle-ci :

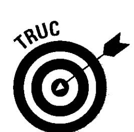
```
public class Pen : IRecordable
{
  public void TakeANote(string sNote)
  {
    // . . . prendre une note sur un bout de papier . . .
  }
}
public class PDA : ElectronicDevice, IRecordable
{
  public void TakeANote(string sNote)
  {
```

```
              nValue);
          throw new Exception(s);
        }
        // commence par donner la valeur 1 à un "accumulateur"
        double dFactorial = 1.0;
        // fait une boucle à partir de nValue en descendant de 1 chaque fois
        // pour multiplier l'accumulateur
        // par la valeur obtenue
        do
        {
          dFactorial *= nValue;
        } while(--nValue > 1);
        // retourne la valeur stockée dans l'accumulateur
        return dFactorial;
      }
    }
    public class Class1
    {
      public static void Main(string[] args)
      {
        try
        {
          // appelle en boucle la fonction Factorial de 6 à -6
          for (int i = 6; i > -6; i--)
          {
            // calcule la factorielle du nombre
            double dFactorial = MyMathFunctions.Factorial(i);
            // affiche le résultat à chaque passage
            Console.WriteLine("i = {0}, factorielle = {1}",
                              i, MyMathFunctions.Factorial(i));
          }
        }
        catch(Exception e)
        {
          Console.WriteLine("Erreur fatale :");
          Console.WriteLine(e.ToString());
        }
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
      }
    }
  }
```

Presque tout le contenu de cette version "exceptionnelle" de `Main()` est placé dans un bloc `try`.

Je peux donc dire que ces trois objets (le stylo, le PDA et l'ordinateur portable) implémentent l'opération TakeANote. En utilisant la magie de l'héritage, je pourrais implémenter la chose en C#, de la façon suivante :

```
abstract class ThingsThatRecord
{
  abstract public void TakeANote(string sNote);
}
public class Pen : ThingsThatRecord
{
  override public void TakeANote(String sNote)
  {
    // . . . prendre une note sur un bout de papier . . .
  }
}
public class PDA : ThingsThatRecord
{
  override public void TakeANote(String sNote)
  {
    // . . . prendre une note sur son PDA . . .
  }
}
public class Laptop : ThingsThatRecord
{
  override public void TakeANote(String sNote)
  {
    // . . . ce que vous voulez . . .
  }
}
```



Si le terme abstract vous remplit de perplexité, revenez au Chapitre 13. Si la notion même d'héritage n'évoque pour vous que mystère, il vous faut passer un peu de temps dans le Chapitre 12.

Cette solution reposant sur l'héritage semble fonctionner très bien pour ce qui ne concerne que l'opération TakeANote(). Une fonction comme RecordTask() peut utiliser la méthode TakeANote() pour noter une liste de commissions sans se soucier du type d'appareil utilisé :

```
void RecordTask(ThingsThatRecord things)
{
  // cette méthode abstraite est implémentée par toutes les classes
  // qui héritent de ThingsThatRecord
  things.TakeANote("Liste de commissions");
  // . . . et ainsi de suite . . .
}
```

- Il mélange le code normal et le code de traitement des erreurs, ce qui obscurcit le chemin d'exécution normal, sans erreur.

Ces problèmes ne paraissent pas si graves dans cet exemple simple, mais ils ne font qu'empirer avec la complexification du code de la fonction appelante. Le résultat est que le code de traitement des erreurs n'est jamais écrit pour traiter autant de conditions d'erreur qu'il devrait.

Heureusement, le mécanisme des exceptions résout tous ces problèmes.

Utiliser un mécanisme d'exceptions pour signaler les erreurs

C# introduit un mécanisme entièrement nouveau, nommé exceptions, pour identifier et traiter les erreurs. Ce mécanisme repose sur les mots-clés `try`, `throw`, `catch`, et `final`. Dans les grandes lignes, il fonctionne de la façon suivante : une fonction va essayer (`try`) d'exécuter une portion de code. Si cette portion de code détecte un problème, elle envoie (`throw`) une indication d'erreur, que la fonction peut attraper (`catch`), et, quoi qu'il arrive, elle exécute finalement (`final`) un certain bloc de code à la fin :

```
public class MyClass
{
  public void SomeFunction()
  {
    // ceci est fait pour attraper une erreur
    try
    {
      // appelle une fonction
      SomeOtherFunction();
      // . . . autant d'autres appels que vous voulez . . .
    }
    catch(Exception e)
    {
      // le contrôle passe par ici en cas
      // d'erreur en un point quelconque du bloc try ou
      // de toute fonction appelée par celui-ci ;
      // l'objet Exception donne la description de l'erreur
    }
  }
  public void SomeOtherFunction()
  {
    // . . . l'erreur se produit quelque part dans la fonction . . .
```

Sceller une classe empêche un autre programme, lequel peut d'ailleurs se trouver quelque part sur Internet, d'en utiliser une version modifiée. Le programme distant peut utiliser la classe telle qu'elle est, mais il ne peut en hériter ni en redéfinir aucun élément.

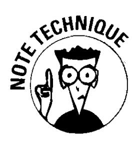
```
i = 6, factorielle = 720
i = 5, factorielle = 120
i = 4, factorielle = 24
i = 3, factorielle = 6
i = 2, factorielle = 2
i = 1, factorielle = 1
i = 0, factorielle = 0
Factorial() a reçu un nombre négatif
Appuyez sur Entrée pour terminer...
```

L'indication d'une condition d'erreur à l'aide d'une valeur retournée par une fonction n'est autre que la manière dont le traitement d'erreur a toujours été réalisé depuis les premiers jours de FORTRAN. Pourquoi changer ?

Je suis là pour signaler ce qui me paraît nécessaire

Quel est l'inconvénient de retourner des codes d'erreur ? C'était très bien pour FORTRAN ! C'est vrai, mais à cette époque les tubes à vide étaient aussi très bien pour les ordinateurs. Malheureusement, l'approche des codes d'erreur présente plusieurs inconvénients.

Tout d'abord, cette solution dépend de la possibilité de retourner une valeur normalement illicite, mais il existe des fonctions pour lesquelles toutes les valeurs qu'il est possible de retourner sont licites. Toutes les fonctions n'ont pas la chance de ne retourner que des valeurs positives. On ne peut pas calculer le logarithme d'un nombre négatif, mais un logarithme peut être positif ou négatif.



Bien que l'on puisse contourner ce problème en utilisant la valeur retournée par une fonction pour une indication d'erreur et un argument de type `out` pour retourner une donnée, cette solution est moins intuitive et fait perdre une partie de la nature expressive d'une fonction. Quoi qu'il en soit, lorsque que vous aurez vu comment fonctionnent les exceptions, vous vous débarrasserez bien vite de cette idée.

D'autre part, un entier ne permet pas de stocker beaucoup d'information. La fonction `Factorial()` retourne -1 si l'argument qui lui est passé est négatif. L'identification du problème pourrait être plus facile si nous savions exactement ce qu'était cette valeur négative, mais il n'y a pas de place pour retourner cette information.

Troisièmement, le traitement des erreurs retournées est optionnel. Vous ne gagnerez pas grand-chose en faisant retourner par `Factorial()` un code

```
      appelle SavingsAccount.Withdraw()
    pour effectuer Test(SpecialSaleAccount)
      appelle SpecialSaleAccount.Withdraw()
  Passage d'un SaleSpecialCustomer
    pour effectuer Test(BankAccount)
      appelle SavingsAccount.Withdraw()
    pour effectuer Test(SavingsAccount)
      appelle SavingsAccount.Withdraw()
    pour effectuer Test(SpecialSaleAccount)
      appelle SaleSpecialCustomer.Withdraw()
    pour effectuer Test(SaleSpecialCustomer)
      appelle SaleSpecialCustomer.Withdraw()
```

Appuyez sur Entrée pour terminer... J'ai mis en gras les appels qui présentent un intérêt particulier. Les classes `BankAccount` et `SavingsAccount` fonctionnent exactement comme on peut l'attendre. Toutefois, en appelant `Test(SavingsAccount)`, `SpecialSalesAccount` et `SaleSpecialCustomer` se passent eux-mêmes comme un `SavingsAccount`. Ce n'est qu'en regardant le niveau immédiatement inférieur que la nouvelle hiérarchie `SaleSpecialCustomer` peut être utilisée à la place de `SpecialSaleAccount`.

Créer une nouvelle hiérarchie

Pourquoi C# permet-il de créer une nouvelle hiérarchie d'héritage ? Le polymorphisme n'est-il pas déjà assez compliqué comme ça ?

C# a été créé pour être un langage "netable", au sens où les classes exécutées par un programme, même les sous-classes, doivent pouvoir être distribuées sur Internet. Autrement dit, un programme que j'écris peut utiliser directement des classes venant de dépôts accessibles par Internet.

Je peux donc étendre une classe que j'ai téléchargée sur Internet. La redéfinition des méthodes d'une hiérarchie de classes standard, testée, peut avoir des effets qui n'étaient pas voulus. L'établissement d'une nouvelle hiérarchie de classes permet à mon programme de bénéficier du polymorphisme sans risque de briser le code existant.

```
public class MyMathFunctions
{
  // ce qui suit représente les valeurs illicites
  public const int NEGATIVE_NUMBER = -1;
  public const int NON_INTEGER_VALUE = -2;
  // Factorial - retourne la factorielle d'une valeur
  //             fournie
  public static double Factorial(double dValue)
  {
    // interdit les nombres négatifs
    if (dValue < 0)
    {
      return NEGATIVE_NUMBER;
    }
    // vérifie si le nombre est bien entier
    int nValue = (int)dValue;
    if (nValue != dValue)
    {
      return NON_INTEGER_VALUE;
    }
    // commence par donner la valeur 1 à un "accumulateur"
    double dFactorial = 1.0;
    // fait une boucle à partir de nValue en descendant de 1 chaque fois
    // pour multiplier l'accumulateur
    // par la valeur obtenue
    do
    {
      dFactorial *= dValue;
      dValue -= 1.0;
    } while(dValue > 1);
    // retourne la valeur stockée dans l'accumulateur
    return dFactorial;
  }
}
public class Class1
{
  public static void Main(string[] args)
  {
    // appelle en boucle la fonction Factorial de 6 à -6
    for (int i = 6; i > -6; i--)
    {
      // calcule la factorielle du nombre
      double dFactorial = MyMathFunctions.Factorial(i);
      if (dFactorial == MyMathFunctions.NEGATIVE_NUMBER)
      {
        Console.WriteLine
                ("Factorial() a reçu un nombre négatif");
        break;
      }
```

```
  {
    Console.WriteLine("  pour effectuer Test(BankAccount)");
    account.Withdraw(100);
  }

  public static void Test2(SavingsAccount account)
  {
    Console.WriteLine("  pour effectuer Test(SavingsAccount)");
    account.Withdraw(100);
  }

  public static void Test3(SpecialSaleAccount account)
  {
    Console.WriteLine("  pour effectuer Test(SpecialSaleAccount)");
    account.Withdraw(100);
  }

  public static void Test4(SaleSpecialCustomer account)
  {
    Console.WriteLine("  pour effectuer Test(SaleSpecialCustomer)");
    account.Withdraw(100);
  }
}
// BankAccount - simule un compte bancaire possédant
//               un numéro de compte (assigné à la création
//               du compte) et un solde
public class BankAccount
{
  // Withdrawal - tout retrait est autorisé jusqu'à la valeur
  //              du solde ; retourne le montant retiré
  virtual public void Withdraw(double dWithdraw)
  {
    Console.WriteLine("    appelle BankAccount.Withdraw()");
  }
}
// SavingsAccount - compte bancaire qui rapporte des intérêts
public class SavingsAccount : BankAccount
{
  override public void Withdraw(double dWithdrawal)
  {
    Console.WriteLine("    appelle SavingsAccount.Withdraw()");
  }
}

// SpecialSaleAccount - compte utilisé uniquement en période de soldes
public class SpecialSaleAccount : SavingsAccount
{
  new virtual public void Withdraw(double dWithdrawal)
```

```
      {
        // appelle en boucle la fonction Factorial de 6 à -6
        for (int i = 6; i > -6; i--)
        {
          // affiche le résultat à chaque passage
          Console.WriteLine("i = {0}, factorielle = {1}",
                            i, MyMathFunctions.Factorial(i));
        }
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
      }
    }
  }
```

La fonction `Factorial()` commence par initialiser à 1 une variable qui va servir d'accumulateur. Elle entre ensuite dans une boucle, multipliant l'accumulateur par des valeurs successivement de plus en plus petites, depuis `nValue` jusqu'à ce que `nValue` atteigne 1. La valeur résultante de l'accumulateur est retournée au point d'appel de la fonction.

L'algorithme de `Factorial()` semble satisfaisant jusqu'à ce que l'on voie comment elle est appelée. `Main()` entre dans une boucle en commençant à la valeur ad hoc pour la factorielle, et décrémente cette valeur à chaque passage, jusqu'à 1. Toutefois, au lieu de s'arrêter, `Main()` continue jusqu'à -6. Je sais bien que -6 est une valeur surprenante, mais il faut bien s'arrêter quelque part.

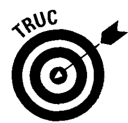
L'exécution de cette fonction produit la sortie suivante :

```
i = 6, factorielle = 720
i = 5, factorielle = 120
i = 4, factorielle = 24
i = 3, factorielle = 6
i = 2, factorielle = 2
i = 1, factorielle = 1
i = 0, factorielle = 0
i = -1, factorielle = -1
i = -2, factorielle = -2
i = -3, factorielle = -3
i = -4, factorielle = -4
i = -5, factorielle = -5
Appuyez sur Entrée pour terminer...
```

Un simple coup d'œil avisé à ces résultats permet de voir qu'ils n'ont aucun sens. Pour commencer, le résultat d'une factorielle ne peut pas être

Ce programme commence par définir la classe AbstractBaseClass avec une seule méthode abstraite, Output(). Comme elle est déclarée abstraite, Output() n'a pas d'implémentation.

Il y a deux classes qui héritent de AbstractBaseClass : SubClass1 et SubClass2. L'une et l'autre sont des classes concrètes, car elles redéfinissent la méthode Output() par des méthodes "réelles".



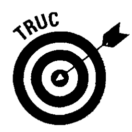
Une classe peut être déclarée abstraite, qu'elle comporte ou non des membres abstraits ; mais une classe ne peut être concrète que lorsque toutes ses méthodes abstraites ont été redéfinies par des méthodes réelles.

Les deux méthodes Output() des deux sous-classes sont différentes, de façon triviale. Toutes deux acceptent une chaîne d'entrée, qu'elles régurgitent vers l'utilisateur. Mais l'une convertit l'ensemble de la chaîne en majuscules, et l'autre convertit l'ensemble en minuscules.

La sortie de ce programme met en évidence la nature polymorphe de la classe AbstractBaseClass :

```
Création d'un objet Subclass1
Appel à SubClass1.Output() depuis TEST

Création d'un objet Subclass2
Appel à SubClass2.Output() depuis test
Appuyez sur Entrée pour terminer...
```



Une classe abstraite est automatiquement virtuelle.

Créer un objet d'une classe abstraite : non !

À propos du programme AbstractInheritance, remarquez qu'il est illicite de créer un objet de la classe AbstractBaseClass, mais que l'argument de Test() est déclaré être un objet de la classe AbstractBaseClass ou de l'une de ses sous-classes. C'est ici la clause concernant les sous-classes qui est cruciale. Les objets de SubClass1 et SubClass2 peuvent être passés, car l'une et l'autre sont des sous-classes concrètes de AbstractBaseClass.

Chapitre 15

Quelques exceptions d'exception

Dans ce chapitre :

Traiter des erreurs avec les codes retournés.

Utiliser plutôt le mécanisme des exceptions.

Créer votre propre classe d'exceptions.

Renvoyer l'envoi de l'extérieur.

Redéfinir des méthodes critiques dans la classe des exceptions.

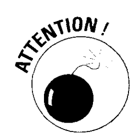
Je sais que c'est difficile à accepter, mais il arrive qu'une méthode (ou une fonction) ne fasse pas ce qu'elle est censée faire. Même celles que j'écris (surtout celles que j'écris) ne font pas toujours ce qu'elles doivent faire. Il est notoire que les utilisateurs aussi ne sont pas fiables. Il suffit que vous demandiez un `int` pour qu'il y en ait un qui entre une chaîne de caractères. Parfois, une méthode poursuit son chemin joyeusement, ignorant voluptueusement qu'elle est en train de produire n'importe quoi. Il y a toutefois de bons programmeurs qui écrivent leurs fonctions de manière à anticiper sur les problèmes et les signaler lorsqu'ils se produisent.



Je parle ici des erreurs à l'exécution, et non des erreurs de compilation que C# vous envoie à la tête lorsque vous essayez de générer votre programme.

Le *mécanisme des exceptions* est un moyen de signaler ces erreurs de la telle manière que la fonction appelante peut comprendre et traiter le problème au mieux.

Le programme commence dans `Main()` par une boucle `while` contenant un bloc `try`. Ce n'est pas rare pour un programme de manipulation de fichiers (dans la section sur `StreamReader`, c'est une approche un peu différente qui aboutit au même résultat).



Placez toutes les fonctions d'I/O dans un bloc `try` avec une instruction `catch` qui génère un message d'erreur approprié. Il est généralement considéré de mauvaise pratique de générer un message d'erreur inapproprié.

La boucle `while` sert à deux choses différentes. Tout d'abord, elle permet au programme de revenir en arrière et d'essayer à nouveau en cas d'échec d'une I/O. Par exemple, si le programme ne trouve pas un fichier que l'utilisateur peut lire, il peut demander à nouveau le nom du fichier pour être sûr de ce qu'il fait avant d'envoyer promener l'utilisateur. Ensuite, l'exécution d'une commande `break` dans le programme vous fait sortir d'un air dégagé du bloc `try`, et vous dépose à la fin de la boucle. C'est un mécanisme très pratique pour sortir d'une fonction ou d'un programme.

Le programme `FileWrite` lit sur la console le nom du fichier à créer. Il se termine en sortant de la boucle `while` si l'utilisateur entre un nom de fichier `null`. L'essentiel du programme se produit dans les deux lignes suivantes.

Pour commencer, le programme crée un objet `FileStream` qui représente le fichier de sortie sur le disque. Le constructeur `FileStream` utilisé ici accepte trois arguments :

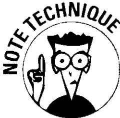
- **Le nom du fichier :** Il est clair que c'est le nom du fichier à ouvrir. Un simple nom de fichier comme `filename.txt` est supposé être dans le répertoire courant. Un nom de fichier qui commence par une barre oblique inverse, comme `\un répertoire\nomdefichier.txt`, est supposé être le chemin d'accès complet au fichier sur la machine locale. Un nom de fichier qui commence par deux barres obliques inverses, par exemple `\\votre machine\un répertoire\un autre répertoire\nomdefichier.txt`, est supposé être le chemin d'accès à un fichier résidant sur une autre machine. À partir de là, le codage du nom de fichier devient rapidement compliqué.

- **Le mode de fichier :** Cet argument spécifie ce que vous voulez faire avec le fichier. Les modes d'écriture de base sont la création (`CreateNew`), l'ajout (`Append`), et la réécriture (`Create`). `CreateNew` crée un nouveau fichier. Si le fichier existe déjà, `CreateNew` envoie une `IOException`. Le mode `Create` simple crée le fichier s'il n'existe pas déjà, mais l'écrase (le remplace) s'il existe déjà. `Append` ajoute quelque chose à la fin d'un fichier s'il existe déjà, et crée un nouveau fichier dans le cas contraire.

fichier que vous avez entré (`Path` est une classe conçue pour manipuler des informations sur les chemins d'accès).



Le *chemin d'accès* (path) est le nom complet du dossier dans lequel se trouve le fichier. Dans le nom de fichier complet `c:\user\temp directory\text.txt`, le chemin d'accès est la partie `c:\user\temp directory`.



La méthode `Combine()` est capable de se rendre compte qu'un fichier comme `c:\test.txt`, `Path()` n'est pas dans le répertoire courant.

En rencontrant la fin de la boucle `while`, soit en exécutant tout le bloc `try`, soit en y étant envoyé par l'instruction `catch`, le programme revient tout en haut pour permettre à l'utilisateur d'écrire un autre fichier.

Voici un exemple d'exécution de ce programme. Ce que j'ai saisi apparaît en gras :

```
Entrez un nom de fichier (Entrez un nom vide pour quitter):TestFile1.txt
Entrez du texte ; une ligne blanche pour arrêter
Je tape quelque chose
Et encore ça
Et puis encore ça

Entrez un nom de fichier (Entrez un nom vide pour quitter):TestFile1.txt
Erreur sur le fichierC:\C#Programs\FileWrite\bin\Debug\TestFile1.txt
Le fichier existe.

Entrez un nom de fichier (Entrez un nom vide pour quitter):TestFile2.txt
Entrez du texte ; une ligne blanche pour arrêter
C'est ici que j'ai fait une erreur. J'aurais dû l'appeler
TestFile2.

Entrez un nom de fichier (Entrez un nom vide pour quitter):
Appuyez sur Entrée pour terminer...
```

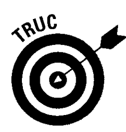
Si j'entre un texte quelconque dans `TestFile1.txt`, tout se passe bien. Mais lorsque que j'essaie d'ouvrir à nouveau le fichier `TestFile1.txt`, le programme m'envoie le message `Le fichier existe`, avec le nom du fichier. Le chemin d'accès au fichier est un peu tourmenté, parce que le "répertoire courant" est celui dans lequel Visual Studio met le fichier exécutable. En corrigeant mon erreur, j'entre du texte en spécifiant le bon nom de fichier (`TestFile2.txt`), sans protestation du programme.

```
                break;
            }
              // erreur envoyée - indique le nom du fichier et l'erreur
            catch(IOException fe)
            {
              Console.WriteLine("{0}\n\n", fe.Message);
            }
          }
          // lit le contenu du fichier
          Console.WriteLine("\nContenu du fichier :");
          try
          {
            // lit une ligne à la fois
            while(true)
            {
              // lit une ligne
              string sInput = sr.ReadLine();
              // quitte si nous n'obtenons rien en retour
              if (sInput == null)
              {
                break;
              }
              // écrit sur la console ce qu'il a lu dans le fichier
              Console.WriteLine(sInput);
            }
          }
          catch(IOException fe)
          {
            // attrape toute erreur de lecture/écriture et la signale
            // (ce qui fait aussi sortir de la boucle)
            Console.Write(fe.Message);
          }
          // ferme le fichier maintenant que nous en avons fini avec lui
          // (en ignorant toute erreur)
          try
          {
            sr.Close();
          }
          catch {}
          // attend confirmation de l'utilisateur
          Console.WriteLine("Appuyez sur Entrée pour terminer...");
          Console.Read();
        }
      }
    }
```

FileRead a une autre approche des noms de fichier. Dans ce programme, l'utilisateur ne lit qu'un fichier. Il doit entrer un nom de fichier valide pour

ReadLine(). Le programme affiche cette ligne sur la console avec l'omniprésent appel Console.WriteLine() avant de revenir au début de la boucle pour lire une autre ligne de texte. L'appel ReadLine() retourne un null lorsque le programme atteint la fin du fichier. Quand cela se produit, le programme sort de la boucle de lecture, ferme l'objet, et se termine.

Remarquez comment l'appel Close() est inséré dans son propre petit bloc try. Une instruction catch sans arguments attrape tout ce qui passe à sa portée. Toute erreur envoyée par Close() est attrapée et ignorée. L'instruction catch est là pour empêcher l'exception de se propager vers le haut de la chaîne et d'arrêter l'exécution du programme. L'erreur est ignorée parce que le programme ne peut rien faire en cas de Close() invalide, et parce qu'il va de toute façon se terminer à la ligne suivante.



Je ne donne l'exemple de catch sans arguments qu'à des fins de démonstration. La présence d'un seul appel dans son propre bloc try avec une instruction catch "attrape-tout" évite qu'un programme s'arrête à cause d'une erreur sans importance. N'utilisez toutefois cette technique que pour une erreur vraiment sans importance, ne pouvant causer aucun dommage.

Voici un exemple d'exécution de ce programme :

```
Entrez le nom d'un fichier texte à lire :TestFilex.txt
Could not find file "C:\C#Programs\FileRead\TestFilex.txt".


Entrez le nom d'un fichier texte à lire :TestFile1.txt

Contenu du fichier :
Je tape quelque chose
Et encore ça
Et puis encore ça
Appuyez sur Entrée pour terminer...
```

Sauf erreur de ma part, c'est la même entrée qu'avec le fichier TestFile1.txt que nous avons créé avec le programme FileWrite.



Ce n'est toutefois pas le même fichier. J'ai dû copier le fichier créé avec TestWrite du répertoire TestWrite\bin\debug au répertoire TestRead\bin\debug. Si vous voulez que ce soit le même fichier dans les deux cas, il vous faut donner le chemin d'accès complet, comme c:\test.txt (j'aurais pu le faire pour ces deux exemples, mais je ne voulais pas mettre de désordre dans votre répertoire racine).

Cinquième partie

Programmer pour Windows avec Visual Studio



"On vient nettoyer le code."

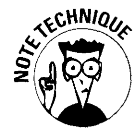
Chapitre 17

Créer une application Windows : le ramage et le plumage

Dans ce chapitre :

- Trouver un problème à résoudre.
- Concevoir une solution.
- Dessiner la solution avec la souris.

Comprendre C# ne suppose pas d'apprendre à écrire des applications Windows pleinement fonctionnelles. Avant de vous mettre à la programmation sous Windows en C#, vous devez avoir de solides notions de la programmation en C#, ce qui ne peut s'acquérir qu'au prix de quelques mois de programmation d'applications console.



Je dois nuancer quelque peu l'affirmation qui précède si vous avez déjà créé des applications Windows dans un langage de programmation comme C++.

Toutefois, vous pouvez vous familiariser avec la programmation pour Windows en passant par les étapes successives de la réalisation d'une application simple. Ce chapitre va vous guider à travers les étapes qui permettent de "dessiner" les applications en utilisant le Concepteur de formulaires de Visual Studio. Le Chapitre 18 présente les étapes qui permettent d'effectuer des opérations suggérées par les formulaires, menus, bannières, boutons, et autres merveilles que vous allez réaliser dans ce chapitre.

2. **Faites une description visuelle de la solution.**

   Tout programme doit être doté d'une interface raisonnablement humaine, faute de quoi un être humain raisonnable ne pourra pas s'interfacer avec lui. Dans le cas d'une application Windows, cela signifie décider des accessoires à utiliser, et où les placer. Choisir les bons accessoires suppose d'avoir au moins fait connaissance avec ceux qui sont disponibles, mais aussi d'avoir un peu de talent artistique (ce qui me met en dehors du coup), et encore d'avoir envie de travailler sur le problème concerné. Pour remplir cette fonction, il vous suffit de vous asseoir devant votre ordinateur. Le Concepteur de formulaires pour Windows est d'une telle souplesse que vous pouvez l'utiliser comme un outil de dessin.

3. **Concevez la solution sur la base de sa présentation et de la description du problème.**

   La conception d'une grande application doit être définie dans les plus grands détails. Par exemple, je travaille en ce moment sur un système de réservation pour une grande compagnie aérienne. Le travail de conception de ce programme occupe quinze personnes pendant à peu près six mois, après quoi le travail de codage et de débogage prend encore douze mois. Cependant, une petite application Windows est souvent largement définie par son interface. C'est plus ou moins le cas avec `SimpleEditor`.

Concevoir la présentation

`SimpleEditor` est un éditeur, et c'est un éditeur simple. Il doit avoir une grande fenêtre dans laquelle l'utilisateur peut entrer du texte. Comme cette fenêtre est la partie la plus importante de n'importe quel éditeur, elle doit occuper pratiquement tout l'écran.

Toute application Windows nécessite un menu Fichier, immédiatement suivi à sa droite par un menu Édition. Les autres éléments de la barre de menus dépendent de l'application, sauf pour l'aide qui en est le dernier.

Dans le menu Fichier, il nous faut un moyen d'ouvrir un fichier (Fichier/Ouvrir), un moyen d'enregistrer un fichier (Fichier/Enregistrer), et un moyen de sortir (Fichier/Quitter). Le petit bouton de fermeture dans le coin supérieur droit de la fenêtre doit avoir le même effet que Fichier/Quitter. Nous n'avons pas besoin d'une commande Fichier/Fermer. C'est très joli, mais comme nous n'en avons pas besoin, c'est une chose que nous pouvons garder pour la version 2.

Comme je l'expliquerai dans les sections qui suivent, ces étapes ne sont pas mal faites si vous les suivez l'une après l'autre.

Créer le cadre de travail de l'application Windows

Pour créer le cadre de travail de l'application Windows :

1. **Sélectionnez Fichier/Nouveau/Projet.**

   La fenêtre Nouveau projet apparaît.

2. **Au lieu de l'icône Application console, cliquez sur l'icône Application Windows, et entrez comme nom** SimpleEditor.



   Dans la fenêtre Nouveau projet, le champ Emplacement spécifie le répertoire dans lequel seront stockés les fichiers de `SimpleEditor`. Autrement dit, Visual Studio va mettre tous les fichiers que je vais créer dans `C:\Programmes C#\SimpleEditor`.

3. **Cliquez sur OK.**

   Visual Studio travaille quelques instants pour générer l'affichage montré par la Figure 17.2.

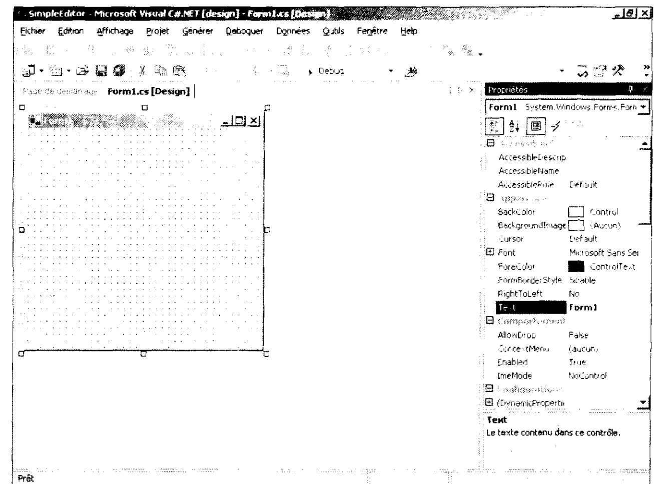


Figure 17.2 : L'affichage initial pour toutes les applications Windows.

Ce curieux affichage s'appelle le Concepteur de formulaires (ou, plus simplement, le Concepteur). Le cadre de vous voyez à gauche est le formulaire, qui va être la base de notre programme `SimpleEditor`.

Ignorez ce type qui se cache derrière le rideau

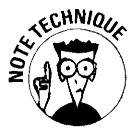
Avant d'aller plus loin, je veux jeter un coup d'œil au code C# généré par le Concepteur. Je veux savoir ce qui se trame là-dedans. L'Explorateur de solutions montre que le fichier source de ce programme se trouve dans un fichier nommé `Form1.cs`, ce qui correspond au nom qui se trouve au-dessus du formulaire dans la fenêtre du Concepteur.

Sélectionnez Affichage/Code pour faire apparaître une nouvelle fenêtre contenant le code source C# de `Form1.cs`, que voici :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace SimpleEditor
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
  {
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;
    public Form1()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }
    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if (components != null)
        {
```

```
            components.Dispose();
          }
        }
        base.Dispose( disposing );
      }
      #region Windows Form Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Name = "Form1";
        this.Text = "Simple Editor";
      }
      #endregion
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
        Application.Run(new Form1());
      }
    }
  }
```

Je sais que le programme doit commencer par `static Main()`, qui se trouve ici tout en bas du listing. Voilà ce qui nourrit ma conviction que c'est ici qu'il faut commencer. La seule instruction que contient `Main()` crée un objet `Form1()` et le passe à une méthode `Application.Run()`. Je ne suis pas sûr de ce que fait `Run()`, mais je soupçonne fortement que la classe `Form1` correspond à la fenêtre Form1 que j'ai vue dans le Concepteur.



En fait, `Application.Run()` lance l'objet `Form` sur son propre thread d'exécution. Le thread initial s'arrête aussitôt que le nouveau `Form1` est créé. Le thread `Form1` se poursuit jusqu'à ce qu'il soit intentionnellement arrêté. Le programme `SimpleEditor` lui-même poursuit son exécution aussi longtemps que des threads définis par l'utilisateur sont actifs.

Le constructeur de `Form1` invoque une méthode `InitializeComponent()`. Tout code d'initialisation du programmeur doit être placé après cet appel (tout au moins, c'est ce que dit le commentaire).



Un formulaire est une fenêtre contenant une barre de titre, et optionnellement des barres de défilement. Dans la terminologie de C#, une fenêtre n'est rien d'autre qu'un cadre rectangulaire dans lequel vous pouvez placer des images ou du texte. Une fenêtre n'a pas nécessairement des menus ou des étiquettes, ni même ces petits boutons Fermer, Réduire et Restaurer.

4. **Générez le programme que Windows vient de créer sur la base du modèle.**

   Vous pouvez me traiter de paranoïaque, mais je veux être certain que toutes les erreurs qui pourront apparaître par la suite seront réellement de mon fait et ne viendront pas de Visual Studio. Sans aucun doute, la solution va se générer sans encombre à ce stade. L'exécution de ce programme ne révèle rien d'autre qu'un formulaire vierge, doté de l'étiquette `Form1`. Il suffit de cliquer sur le bouton Fermer pour arrêter le programme.

   Le volet qui occupe la partie droite de l'affichage est la fenêtre Propriétés. Ça ne saute peut-être pas aux yeux, mais son contenu est en relation directe avec le formulaire qui est dans la partie gauche de l'affichage. Par exemple, vous pouvez voir que la propriété `Text` est `Form1`. Vous pouvez la modifier pour vous rendre compte de l'effet produit.

5. **Sélectionnez la propriété** `Text`**, et donnez-lui la valeur** Simple Editor.

   L'étiquette `Form1` contenue dans la barre de titre du formulaire devient Simple Editor.

6. **Générez à nouveau l'application, et exécutez-la.**

   Le nom du formulaire a changé, comme le montre la Figure 17.3.

SimpleEditor

Figure 17.3 : Changer la propriété `Text` du formulaire change le nom qui apparaît dans sa barre de titre.

Le menu Édition a besoin des trois grandes options d'édition : Couper, Copier et Coller. D'autre part, tous les éditeurs comprennent les raccourcis clavier de ces trois options : Ctrl+X, Ctrl+C, et Ctrl+V, respectivement.

`SimpleEditor` aura également besoin d'un menu Format, comportant les options Gras et Italique pour mettre en forme le texte.

Fournir une aide véritable est une tâche difficile – beaucoup trop compliquée pour un éditeur simple comme `SimpleEditor`. Le menu d'aide de cette application devra se contenter du minimum absolu : l'option À propos de.

Dernière exigence : il nous faut un moyen de contrôler la taille de police. Voilà une chose qui laisse la place à un peu de fantaisie. En plus d'une simple fenêtre dans laquelle l'utilisateur peut entrer la taille de police souhaitée, `SimpleEditor` y ajoutera une sorte de barre munie d'un index que l'on peut faire glisser, que nous appellerons `TrackBar`. Pour obtenir 8 points, faites glisser l'index à l'extrémité gauche. Faites-le glisser à l'extrémité droite, et vous obtenez 24 points. (J'ai une autre raison de procéder ainsi : je veux vous montrer comment relier deux objets d'I/O de manière qu'un changement dans l'un soit répercuté dans l'autre.)

Ma solution

Avec les paramètres que j'ai décrits dans la section précédente, je suis arrivé à la solution montrée par la Figure 17.1. Vos propres résultats peuvent être différents selon vos goûts personnels.

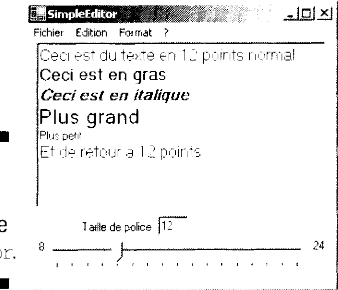


Figure 17.1 : Ma solution du problème `SimpleEditor`.

Dessiner la solution

Comme vous pouvez l'imaginer, j'ai dû passer par de nombreuses étapes pour arriver en partant de zéro à l'œuvre d'art montrée par la Figure 17.1.

Quel est le problème ?

Il m'a fallu une longue et difficile réflexion (au moins un quart d'heure) pour imaginer un problème qui mette en lumière la puissance de C# sans me faire prendre du poids. Le voici : créer un éditeur simple que nous appellerons `SimpleEditor`. Il aura les caractéristiques suivantes :

- L'utilisateur peut entrer et effacer du texte (sinon, ce ne serait pas vraiment un éditeur).
- L'utilisateur peut couper et coller du texte, non seulement dans `SimpleEditor`, mais aussi entre `SimpleEditor` et d'autres applications, par exemple Word.
- `SimpleEditor` supporte les polices en gras, en italique ou les deux.
- L'utilisateur peut sélectionner une taille de police de 8 à 24 points. Ces limites sont arbitraires, mais il s'agit ici de ne pas aller trop loin en nombre de points.
- `SimpleEditor` ne doit pas vous permettre de quitter sans vous avoir demandé poliment d'enregistrer le fichier que vous venez de modifier (mais vous restez libre de quitter sans enregistrer si c'est bien ce que vous voulez).

Exposer le problème

Chaque fois que vous êtes devant un problème à résoudre, vous devez commencer par vous mettre devant le tableau noir et réfléchir sérieusement aux obstacles à franchir. Dans le cas d'une application Windows, cette tâche se divise en trois étapes :

1. **Décrivez le problème en détail.**

   Ces détails sont les spécifications auxquelles doit se conformer l'application. Au cours de la programmation, vous pourrez être tenté d'ajouter une fonctionnalité ici ou là. Résistez. Cette maladie s'appelle *fonctionnalite*. Tout en avançant, notez les améliorations possibles pour une version future, mais l'ajout de fonctionnalités en cours de route fait courir le risque de créer une application qui finit par être tout à fait autre chose que ce qu'elle était censée être au départ.

Dans cette partie...

Comprendre C# est une chose, apprendre à écrire une application Windows complète avec tous ses assemblages et ses décorations bien en place en est une autre. Rien que pour le plaisir, la cinquième partie vous guide pas à pas dans l'utilisation de C# avec l'interface Visual Studio afin de créer une application Windows "qui ne soit pas triviale". Vous serez fier du résultat, même si vos enfants n'appellent pas leurs copains pour le voir.

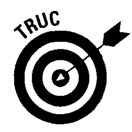
que le programme donne la sortie attendue. Une fois que le programme a lu le fichier, il se termine. Si l'utilisateur veut lire un autre fichier, il lui suffit d'exécuter à nouveau le programme.

Le programme commence par une boucle `while`, comme son cousin `FileWrite`. Dans cette boucle, il va chercher le nom de fichier entré par l'utilisateur. Si le nom de fichier est vide, le programme envoie un message d'erreur : `Vous avez entré un nom de fichier vide`. Dans le cas contraire, le nom de fichier est utilisé pour ouvrir un objet `FileStream` en mode de lecture. L'appel `File.Open()` est ici le même que celui utilisé dans `FileWrite` :

- Le premier argument est le nom du fichier.
- Le deuxième argument est le modèle du fichier. Le mode `FileMode.Open` dit : "Ouvrir le fichier s'il existe, sinon envoyer une exception." L'autre possibilité est `OpenNew`, qui crée un fichier de longueur nulle si celui-ci n'existe pas déjà. Personnellement, je n'ai jamais rencontré le besoin de ce mode (qui veut lire un fichier vide ?), mais chacun mène sa barque comme il l'entend.
- Le dernier argument indique que je veux lire à partir de ce `FileStream`. Les autres solutions sont `Write` et `ReadWrite`.

L'objet `FileStream fs` résultant est alors inséré dans un objet `StreamReader sr` qui offre des méthodes pratiques pour accéder au fichier texte.

Toute cette section d'ouverture de fichier est enchâssée dans un bloc `try`, lui-même enchâssé dans une boucle `while`, insérée dans une énigme. Ce bloc `try` est strictement réservé à l'ouverture de fichier. Si une erreur se produit pendant le processus d'ouverture, l'exception est attrapée, un message d'erreur est affiché, et le programme reprend au début de la boucle pour demander à nouveau un nom de fichier à l'utilisateur. Toutefois, si le processus aboutit à un objet nouveau-né `StreamReader` en bonne santé, la commande `break` fait sortir de la logique d'ouverture de fichier et fait passer le chemin d'exécution du programme à la section de lecture.

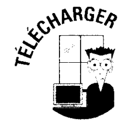


`FileRead` et `FileWrite` représentent deux manières différentes de traiter des exceptions de fichier. Vous pouvez insérer tout le programme de traitement de fichier dans un même bloc `try`, comme dans `FileWrite`, ou bien vous pouvez donner son propre bloc `try` à la section d'ouverture de fichier. Cette dernière solution est généralement la plus facile, et elle permet de générer un message d'erreur plus précis.

Une fois le processus d'ouverture de fichier terminé, le programme `FileRead` lit une ligne de texte dans le fichier en utilisant l'appel

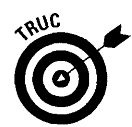
Améliorez votre compréhension et votre vitesse de lecture avec StreamReader

Il est très agréable d'écrire sur un fichier, mais c'est plutôt inutile si vous ne pouvez pas lire le fichier par la suite. Le programme FileRead suivant affiche sur la console ce qu'il lit dans le fichier. Ce programme lit un fichier texte comme celui que crée FileWrite :



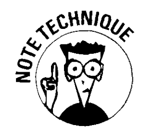

```
// FileRead - lit un fichier texte et l'écrit
//           sur la console
using System;
using System.IO;
namespace FileRead
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // il nous faut un objet pour lire le fichier
      StreamReader sr;
      string sFileName = "";
      // continue à essayer de lire un nom de fichier jusqu'à ce qu'il en
      // trouve un (la seule manière de quitter pour l'utilisateur est
      // d'arrêter le programme en appuyant sur Ctrl + C)
      while(true)
      {
        try
        {
          // lit le nom du fichier d'entrée
          Console.Write("Entrez le nom d'un fichier texte à lire :");
          sFileName = Console.ReadLine();
          // l'utilisateur n'a rien entré ; envoie une erreur
          // pour lui dire que ce n'est pas satisfaisant
          if (sFileName.Length == 0)
          {
            throw new IOException("Vous avez entré un nom de fichier vide");
          }
          // ouvre un flux de fichier pour la lecture ; ne crée pas
          // le fichier s'il n'existe pas déjà
          FileStream fs = File.Open(sFileName,
            FileMode.Open,
            FileAccess.Read);
          // convertit ceci en StreamReader - ce sont les trois premiers
          // octets du fichier qui seront utilisés pour indiquer
          // l'encodage utilisé (mais pas le langage)
          sr = new StreamReader(fs, true);
```

- **Le type d'accès :** Un fichier peut être ouvert pour la lecture, l'écriture ou les deux.



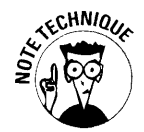
FileStream dispose de nombreux constructeurs, dont chacun correspond par défaut à un ou deux des arguments de mode et d'accès. Toutefois, à mon humble avis, il vaut mieux spécifier explicitement ces arguments, car ils ont un effet important sur le programme.

Dans la ligne suivante, le programme insère dans un objet StreamWriter, sw, l'objet FileStream qu'il vient d'ouvrir. La classe StreamWriter permet d'insérer les objets FileStream, afin de fournir un ensemble de méthodes pour traiter du texte. Le premier argument du constructeur StreamWriter est l'objet FileStream. Le deuxième spécifie le type d'encodage à utiliser. L'encodage par défaut est UTF8.



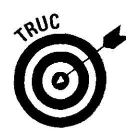
Il n'est pas nécessaire de spécifier l'encodage pour lire un fichier. StreamWriter inscrit le type d'encodage dans les trois premiers octets du fichier. À l'ouverture du fichier, ces trois octets sont lus pour déterminer l'encodage.

Le programme FileWrite commence alors à lire sous forme de chaînes les lignes saisies sur la console. Le programme arrête de lire lorsque l'utilisateur entre une ligne blanche, mais jusque-là il continue à absorber tout ce qu'on lui donne pour le déverser dans l'objet StreamWriter sw en utilisant la méthode WriteLine().



La similitude entre StreamWriter.WriteLine() et Console.WriteLine() n'est pas qu'une coïncidence.

Enfin, le fichier est fermé par l'instruction sw.Close().

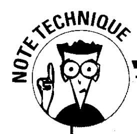


Remarquez que le programme donne à la référence sw la valeur null à la fermeture du fichier. Un objet fichier est parfaitement inutile une fois que celui-ci a été fermé. Il est de bonne pratique de donner à la référence la valeur null une fois qu'elle est devenue invalide, afin de ne pas essayer de l'utiliser à nouveau dans l'avenir.

Le bloc catch qui suit la fermeture du fichier est un peu comme un gardien de but : il est là pour attraper toute erreur de fichier qui aurait pu se produire en un endroit quelconque du programme. Ce bloc émet un message d'erreur, contenant le nom du fichier qui en est responsable. Mais il ne se contente pas d'indiquer simplement le nom du fichier : il vous donne son chemin d'accès complet, en ajoutant à l'aide de la méthode Path.Combine() le nom du répertoire courant avant le nom de

```
                //                  FileAccess.Write,
                //                  FileAccess.ReadWrite
                FileStream fs = File.Open(sFileName,
                                          FileMode.CreateNew,
                                          FileAccess.Write);
                // génère un flux de fichier avec des caractères UTF8
                sw = new StreamWriter(fs, System.Text.Encoding.UTF8);
                // lit une chaîne à la fois, et envoie chacune au
                // FileStream ouvert pour écriture
                Console.WriteLine("Entrez du texte ; ligne blanche pour arrêter");
                while(true)
                {
                  // lit la ligne suivante sur la console ;
                  // quitte si la ligne est blanche
                  string sInput = Console.ReadLine();
                  if (sInput.Length == 0)
                  {
                    break;
                  }
                  // écrit sur le fichier de sortie la ligne qui vient d'être lue
                  sw.WriteLine(sInput);
                }
                // ferme le fichier que nous avons créé
                sw.Close();
                sw = null;
              }
              catch(IOException fe)
              {
                // une erreur s'est produite quelque part pendant
                // le traitement du fichier - indique à l'utilisateur
                // le nom complet du fichier :
                // ajoute au nom du répertoire par défaut
                // celui du fichier
                string sDir = Directory.GetCurrentDirectory();
                string s = Path.Combine(sDir, sFileName);
                Console.WriteLine("Erreur sur le fichier{0}", s);
                // affiche maintenant le message d'erreur de l'exception
                Console.WriteLine(fe.Message);
              }
            }
            // attend confirmation de l'utilisateur
            Console.WriteLine("Appuyez sur Entrée pour terminer...");
            Console.Read();
          }
        }
      }
```

`FileWrite` utilise l'espace de nom `System.IO` ainsi que `System`. `System.IO` contient les fonctions d'I/O sur les fichiers.



I/O asynchrones : est-ce que ça vaut la peine d'attendre ?

Normalement, un programme attend qu'une requête d'I/O sur un fichier soit satisfaite avant de poursuivre son exécution. Appelez une méthode `read()`, et vous ne récupérerez généralement pas le contrôle aussi longtemps que les données du fichier ne seront pas installées à bord en sécurité. C'est ce que l'on appelle une *I/O synchrone*.

Avec C#, les classes de `System.IO` supportent également les I/O asynchrones. En les utilisant, l'appel à `read()` restitue immédiatement le contrôle pour permettre au programme de poursuivre son exécution pendant que la requête d'I/O est satisfaite à l'arrière-plan. Le programme est libre de vérifier l'état d'un indicateur pour savoir si la requête d'I/O a abouti.

C'est un peu comme de faire cuire un hamburger. Avec des I/O synchrones vous mettez la viande hachée à cuire sur la plaque chauffante, vous la surveillez jusqu'à ce qu'elle soit cuite, et c'est seulement à partir de là que vous pouvez vous mettre à couper les oignons qui vont aller dessus.

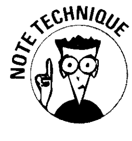
Avec des I/O asynchrones, vous pouvez couper les oignons pendant que la viande hachée est en train de cuire. De temps en temps, vous jetez un coup d'œil pour voir si elle est cuite. Le moment venu, vous abandonnez un instant vos oignons, et vous prenez la viande sur la plaque chauffante pour la mettre sur le pain.

Les I/O asynchrones peuvent améliorer significativement les performances d'un programme, mais elles ajoutent un niveau supplémentaire de complexité.

Utiliser `StreamWriter`

Les programmes génèrent deux sortes de sortie. Certains programmes écrivent des blocs de données dans un pur format binaire. Ce type de sortie est utile pour stocker des objets d'une manière efficace.

Beaucoup de programmes, sinon la plupart, lisent et écrivent des chaînes de texte, lisibles par un être humain. Les classes de flux `StreamWriter` et `StreamReader` sont les plus souples des classes accueillantes pour l'homme.

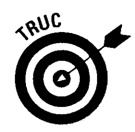


Les données lisibles par un être humain étaient antérieurement des chaînes ASCII, ou, un peu plus tard, ANSI. Ces deux sigles se réfèrent aux organisations de standardisation qui ont défini ces formats. Toutefois, le codage ANSI ne permet pas d'intégrer les alphabets venant de plus loin que l'Autriche à l'Est, et de plus loin que Hawaï à l'Ouest. Il ne peut contenir que l'alphabet latin. Il ne dispose pas de l'alphabet cyrillique, hébreu,

- Une méthode déclarée internal est accessible par toutes les classes du même espace de nom. Aussi, l'appel class2.D_internal() n'est pas autorisé. L'appel class3.C_internal() est autorisé parce que Class3 fait partie de l'espace de nom AccessControl.
- Le mot-clé internal protected combine l'accès internal et l'accès protected. Aussi, l'appel class1.E_internalprotected() est autorisé, parce que Class1 étend Class2 (c'est la partie protected). L'appel class3.E_internalprotected() est également autorisé, parce que Class1 et Class3 font partie du même espace de nom (c'est la partie internal).
- La déclaration de Class3 comme internal a pour effet de réduire l'accès à celle-ci à internal, ou moins. Aussi, les méthodes public deviennent internal, alors que les méthodes protected deviennent internal protected.

Ce programme donne la sortie suivante :

```
Class2.A_public
Class2.B_protected
Class1.C_private
Class3.D_internal
Class2.E_internalprotected
Class3.E_internalprotected
Appuyez sur Entrée pour terminer...
```



Déclarez toujours les méthodes avec un accès aussi restreint que possible. Une méthode privée peut être modifiée à volonté sans inquiéter de l'effet que cela pourrait avoir sur d'autres classes. Une classe ou une méthode interne de MathRoutines est utilisé par d'autres classes de nature mathématique. Si vous n'êtes pas convaincu de la sagesse du couplage faible entre les classes, allez voir le Chapitre 11.

Rassembler des données dans des fichiers

Les applications console de ce livre reçoivent essentiellement leurs entrées de la console, et y envoient de même leur sortie. Les programmes des autres sections que celle-ci ont mieux à faire (ou autre chose) que de vous embêter avec des manipulations de fichiers. Je ne veux pas les obscurcir avec la question supplémentaire des entrées/sorties (I/O). Toutefois, les applications console qui n'effectuent pas d'opération d'entrée/sortie sur des fichiers sont à peu près aussi courantes que les phoques dans la Seine.

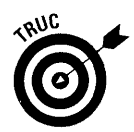
```
      // la même classe
      //class2.C_private();
      class1.C_private();
      // les méthodes internes ne sont accessibles que par
      // les classes du même espace de nom
      //class2.D_internal();
      class3.D_internal();
      // les méthodes internes protégées sont accessibles
      // soit par la hiérarchie d'héritage soit par
      // toute classe du même espace de nom
      class1.E_internalprotected();
      class3.E_internalprotected();
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
    public void C_private()
    {
      Console.WriteLine("Class1.C_private");
    }
  }
  // Class3 - une classe interne est accessible aux autres
  //          classes du même espace de nom, mais
  //          pas aux classes externes qui utilisent cet
  //          espace de nom
  internal class Class3
  {
    // la déclaration d'une classe comme interne force toutes
    // les méthodes publiques à être également internes
    public void A_public()
    {
      Console.WriteLine("Class3.A_public");
    }
    protected void B_protected()
    {
      Console.WriteLine("Class3.B_protected");
    }
    internal void D_internal()
    {
      Console.WriteLine("Class3.D_internal");
    }
    public void E_internalprotected()
    {
      Console.WriteLine("Class3.E_internalprotected");
    }
  }
}
```

Utiliser un espace de nom avec le mot-clé `using`

Se référer à une classe par son nom pleinement qualifié peut devenir un peu fastidieux. Le mot-clé `using` de C# vous permet d'éviter ce pensum. La commande `using` ajoute l'espace de nom spécifié à une liste d'espaces de nom par défaut que C# consulte pour essayer de résoudre un nom de classe. L'exemple de programme suivant se compile sans une plainte :

```
namespace Paint
{
  public class PaintColor
  {
    public PaintColor(int nRed, int nGreen, int nBlue) {}
    public void Paint() {}
    public static void StaticPaint() {}
  }
}
namespace MathRoutines
{
  // ajoute Paint aux espaces de nom dans lesquels on cherche
  // automatiquement
  using Paint;
  public class Test
  {
    static public void Main(string[] args)
    {
      // crée un objet dans un autre espace de nom - il n'est
      // pas nécessaire de faire figurer le nom de l'espace de nom, car
      // celui-ci est inclus dans une instruction "using"
      PaintColor black = new PaintColor(0, 0, 0);
      black.Paint();
      PaintColor.StaticPaint();
    }
  }
}
```

La commande `using` dit : "Si vous ne trouvez pas la classe spécifiée dans l'espace de nom courant, voyez si vous la trouvez dans celui-ci." Vous pouvez spécifier autant d'espaces de nom que vous voulez, mais toutes les commandes `using` doivent apparaître l'une après l'autre tout à fait au début du programme.



Tous les programmes commencent par la commande `using System;`. Elle donne au programme un accès automatique à toutes les fonctions de la bibliothèque système, comme `WriteLine()`.

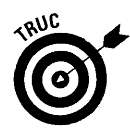
TranslationLibrary respectivement à ces deux ensembles de classes évite le problème : FileIO.Convert ne peut pas être confondu avec TranslationLibrary.Convert.

Déclarer un espace de nom

On déclare un espace de nom en utilisant le mot-clé namespace, suivi par un nom et un bloc d'accolades ouvrante et fermante. Les classes spécifiées dans ce bloc font partie de l'espace de nom.

```
namespace MyStuff
{
  class MyClass {}
  class UrClass {}
}
```

Dans cet exemple, MyClass et UrClass font partie de l'espace de nom MyStuff.



L'Assistant Application de Visual Studio place chaque classe qu'il crée dans un espace de nom portant le même nom que le répertoire qu'il crée. Examinez tous les programmes de ce livre : ils ont tous étés créés à l'aide de l'Assistant Application. Par exemple, le programme AlignOutput a été créé dans le dossier AlignOutput. Le nom du fichier source est Class1.cs, qui correspond au nom de la classe par défaut. Le nom de l'espace de nom dans lequel se trouve Class1.cs est le même que celui du dossier : AlignOutput.



Si vous ne spécifiez pas une désignation d'espace de nom, C# place votre classe dans l'espace de nom global. C'est l'espace de nom de base pour tous les autres espaces de nom.

Accéder à des modules du même espace de nom

Le nom de l'espace de nom d'une classe est une partie du nom de la classe étendue. Voyez l'exemple suivant :

```
namespace MathRoutines
{
  class Sort
  {
    public void SomeFunction(){}
```

ne peut pas être modifié par deux programmeurs en même temps. Chacun d'eux a besoin de son propre fichier source. Enfin, la compilation d'un module de grande taille peut prendre beaucoup de temps (on peut toujours aller prendre un café, mais il arrive un moment où votre patron devient soupçonneux). Recompiler un tel module parce qu'une seule ligne d'une seule classe a été modifiée devient intolérable.

Pour toutes ces raisons, un bon programmeur C# divise son programme en plusieurs fichiers source .CS, qui sont compilés et générés ensemble afin de former un seul exécutable.

Imaginez un système de réservation de billets d'avion : il y a l'interface avec les agents de réservation que les clients appellent au téléphone, une autre interface pour la personne qui est au comptoir d'enregistrement, la partie Internet, sans parler de la partie qui vérifie l'occupation des sièges dans l'avion, plus la partie qui calcule le prix (y compris les taxes), et ainsi de suite. Un programme comme celui-ci devient énorme bien avant d'être terminé.

Rassembler toutes ces classes dans un même fichier source `Class1.cs` est remarquablement déraisonnable, pour les raisons suivantes :

- Un fichier source ne peut être modifié que par une seule personne à la fois. Vous pouvez avoir vingt à trente programmeurs travaillant en même temps sur un grand projet. Un seul fichier pour vingt-quatre programmeurs impliquerait que chacun d'eux ne pourrait travailler qu'une heure par jour, à supposer qu'ils se relaient vingt-quatre heures sur vingt-quatre. Si vous divisiez le programme en vingt-quatre fichiers, il serait possible, bien que difficile, que tous les programmeurs travaillent en même temps. Mais si vous divisez le programme de telle manière que chaque classe a son propre fichier, l'orchestration du travail de ces vingt-quatre programmeurs devient beaucoup plus facile.

- Un fichier source unique peut devenir extrêmement difficile à comprendre. Il est beaucoup plus aisé de saisir le contenu d'un module comme `ResAgentInterface.cs`, `GateAgentInterface.cs`, `ResAgent.cs`, `GateAgent.cs`, `Fare.cs` ou `Aircraft.cs`.

- La régénération complète d'un grand programme comme un système de réservation de billets d'avion peut prendre beaucoup de temps. Vous n'aurez certainement pas envie de régénérer toutes les instructions qui composent le système simplement parce qu'un programmeur a modifié une seule ligne. Avec un programme divisé en plusieurs fichiers, Visual Studio peut régénérer uniquement le fichier modifié, et rassembler ensuite tous les fichiers objet.

Le programme CustomException donne la sortie suivante :

```
Erreur fatale inconnue :
Le message est <Impossible d'inverser 0>, l'objet est (Value = 0)
CustomException.MathClass
Exception envoyée parDouble Inverse()
Appuyez sur Entrée pour terminer...
```

Jetons un coup d'œil à cette sortie : le message Erreur fatale inconnue : vient de Main(). La chaîne Le message est <Impossible d'inverser 0>, l'objet est <~~> vient de CustomException. Le message Value = 0 vient de l'objet MathClass lui-même. La dernière ligne, Exception envoyée parDouble Inverse, vient de CustomException.

ToString(), la carte de visite de la classe

Toutes les classes héritent d'une classe de base commune, judicieusement nommé Object. C'est au Chapitre 17 que j'explore cette propriété qui unifie les classes, mais il est utile de mentionner ici que Object contient une méthode, ToString(), qui convertit en string le contenu de l'objet. L'idée est que chaque classe doit redéfinir la méthode ToString() par une méthode lui permettant de s'afficher elle-même d'une façon pertinente. Dans le chapitre précédent, j'ai utilisé la méthode GetString() parce que je ne voulais pas y aborder les questions d'héritage, mais le principe est le même. Par exemple, une méthode Student.ToString() pourrait afficher le nom et le numéro d'identification de l'étudiant.

La plupart des fonctions, même les fonctions intégrées de la bibliothèque C#, utilisent la méthode ToString() pour afficher des objets. Ainsi, le remplacement de ToString() a pour effet secondaire très utile que l'objet sera affiché dans son propre format, quelle que soit la fonction qui se charge de l'affichage.

Comme dirait Bill Gates, "C'est cool."

```
    //           le message standard Exception.ToString()
    override public string ToString()
    {
      string s = Message + "\n";
      s += base.ToString();
      return s;
    }
    // Inverse - retourne 1/x
    public double Inverse()
    {
      if (nValueOfObject == 0)
      {
        throw new CustomException("Impossible d'inverser 0", this);
      }
      return 1.0 / (double)nValueOfObject;
    }
}
  public class Class1
  {
    public static void Main(string[] args)
    {
      try
      {
        // prend l'inverse de 0
        MathClass mathObject = new MathClass("Valeur", 0);
        Console.WriteLine("L'inverse de d.Value est{0}",
                          mathObject.Inverse());
      }
      catch(Exception e)
      {
        Console.WriteLine("\nErreur fatale inconnue :\n{0}",
                          e.ToString());
      }

      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

Permettez-moi de faire une remarque : cette classe `CustomException` n'est pas si remarquable que cela. Elle stocke un message et un objet, tout comme `MyException`. Toutefois, au lieu de fournir de nouvelles méthodes pour accéder à ces données, elle remplace la propriété `Message` existante qui retourne le message d'erreur contenu dans l'exception, et la méthode `ToString()` qui retourne le message plus l'indication de pile.

Renvoyer le même objet exception présente un avantage et un inconvénient. Cela permet aux fonctions intermédiaires d'attraper des exceptions pour libérer ou fermer des éléments alloués par elles, tout en permettant à l'utilisateur final de l'objet exception de suivre l'indication de pile jusqu'à la source de l'exception. Toutefois, une fonction intermédiaire ne peut pas (ou ne doit pas) ajouter des informations supplémentaires à l'exception en la modifiant avant de la renvoyer.

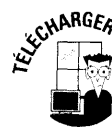
Redéfinir une classe d'exceptions

La classe d'exceptions suivante définie par l'utilisateur peut stocker des informations supplémentaires qui ne pourraient pas l'être dans un objet `Exception` conventionnel :

```
// MyException - ajoute à la classe standard Exception
//              une référence à MyClass
public class MyException : Exception
{
  private MyClasss myobject;
  MyException(string sMsg, MyClass mo) : base(sMsg)
  {
    myobject = mo;
  }
  // permet aux classes extérieures d'accéder à une classe d'information
  public MyClass MyObject{ get {return myobject;}}
}
```

Voyez à nouveau ma bibliothèque de fonctions `BrilliantLibrary`. Ces fonctions savent comment remplir ces nouveaux membres de la classe `MyException` et aller les chercher, fournissant ainsi uniquement les informations nécessaires pour remonter à la source de toute erreur connue et de quelques autres restant à découvrir. L'inconvénient de cette approche est que seules les fonctions de la bibliothèque `BrilliantLibrary` peuvent recevoir un bénéfice quelconque des nouveaux membres de `MyException`.

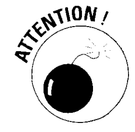
Le remplacement des méthodes déjà présentes dans la classe `Exception` peut donner des fonctions existantes autres que l'accès `BrilliantLibrary` aux nouvelles données. Considérez la classe d'exceptions définie dans le programme `CustomException` suivant :




```
// CustomException - crée une exception personnalisée qui
//                   affiche les informations que nous voulons, mais
//                   dans un format plus agréable
```

types d'exception définie pour la brillante bibliothèque de classe que je viens d'écrire (c'est pour ça que je l'appelle BrilliantLibrary). Les fonctions qui composent BrilliantLibrary envoient et attrapent des exceptions MyException.

Toutefois, les fonctions de la bibliothèque BrilliantLibrary peuvent aussi appeler des fonctions de la bibliothèque générique System. Les premières peuvent ne pas savoir comment traiter les exceptions de la bibliothèque System, en particulier si elles sont causées par une entrée erronée.



Si vous ne savez pas quoi faire avec une exception, laissez-la passer pour qu'elle arrive à la fonction appelante. Mais soyez honnête avec vous-même : ne laissez pas passer une exception parce que vous n'avez simplement pas le courage d'écrire le code de traitement d'erreur correspondant.

Relancer un objet

Dans certains cas, une méthode ne peut pas traiter entièrement une erreur, mais ne veut pas laisser passer l'exception sans y mettre son grain de sel. C'est comme une fonction mathématique qui appelle Factorial() pour s'apercevoir qu'elle renvoie une exception. Même si la cause première du problème peut être une donnée incorrecte, la fonction mathématique est peut-être en mesure de fournir des indications supplémentaires sur ce qui s'est passé.

Un bloc catch peut digérer partiellement l'exception envoyée et ignorer le reste. Ce n'est pas ce qu'il y a de plus beau, mais ça existe.

L'interception d'une exception d'erreur est une chose très courante pour les méthodes qui allouent des éléments. Par exemple, imaginez une méthode F() qui ouvre un fichier quand elle est invoquée, et le referme quand elle se termine. Quelque part dans le cours de son exécution, F() invoque G(). Une exception envoyée de G() passerait directement à travers F() sans lui laisser la moindre chance de fermer le fichier. Celui-ci resterait donc ouvert jusqu'à ce que le programme lui-même se termine. Une solution idéale serait que F() contienne un bloc catch qui ferme les fichiers ouverts. Bien entendu, F() est libre de passer l'exception au niveau supérieur après en avoir fait ce qu'il fallait pour ce qui la concerne.

Il y a deux manières de renvoyer une erreur. La première consiste à envoyer une deuxième exception, avec les mêmes informations ou éventuellement des informations supplémentaires :

```
      Console.WriteLine(e.Message);
    }
  }

  // f2 -  - préparez-vous à attraper une exception MyException
  public void f2(bool bExceptionType)
  {
    try
    {
      f3(bExceptionType);
    }
    catch(MyException me)
    {
      Console.WriteLine("Exception MyException attrapée dans f2()");
      Console.WriteLine(me.Message);
    }
  }

  // f3 -  - n'essayez pas d'attraper des exceptions
  public void f3(bool bExceptionType)
  {
    f4(bExceptionType);
  }

  // f4 -  - envoie des exceptions d'un type ou d'un autre
  public void f4(bool bExceptionType)
  {
    // nous travaillons avec un objet local
    MyClass mc = new MyClass();
    if(bExceptionType)
    {
      // une erreur se produit - l'objet est envoyé avec l'exception
      throw new MyException("MyException envoyée dans f4()",
                            mc);
    }
    throw new Exception("Exception générique envoyée dans f4()");
  }

  public static void Main(string[] args)
  {
    // envoie d'abord une exception générique
    Console.WriteLine("Envoie d'abord une exception générique");
    new Class1().f1(false);
    // envoie maintenant une de mes exceptions
    Console.WriteLine("\nEnvoie d'abord une exception spécifique");
    new Class1().f1(true);

    // attend confirmation de l'utilisateur
    Console.WriteLine("Hit Appuyez sur Entrée pour terminer...");
```

```
    }
    catch(Exception e)
    {
      // les autres exceptions non encore attrapées sont attrapées ici
    }
  }
```

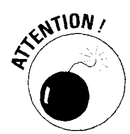
Si `SomeOtherFunction()` envoyait un objet `Exception`, celui-ci ne serait pas attrapé par l'instruction `catch(MyException)` car une `Exception` n'est pas de type `MyException`. Il serait attrapé par l'instruction `catch` suivante : `catch(Exception)`.



Toute classe qui hérite de `MyException` EST_UNE `MyException` :

```
class MySpecialException : MyException
{
  // . . . instructions quelconques ...
}
```

Si elle en a la possibilité, l'instruction `catch MyException` attrapera tout objet `MySpecialException` envoyé.



Faites toujours se succéder les instructions `catch` de la plus spécifique à la plus générale. Ne placez jamais en premier l'instruction `catch` la plus générale :

```
public void SomeFunction()
{
  try
  {
    SomeOtherFunction();
  }
  catch(Exception me)
  {
    // tous les objets MyException sont attrapés ici
  }
  catch(MyException e)
  {
    // aucune exception ne parvient jamais jusqu'ici parce qu'elle
    // est attrapée par une instruction catch plus générale
  }
}
```

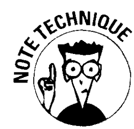
Dans cet exemple, l'instruction `catch` la plus générale coupe l'herbe sous le pied de la suivante en interceptant tous les envois.

Cette classe CustomException est faite sur mesure pour signaler une erreur au logiciel qui traite avec la tristement célèbre MyClass. Cette sous-classe d'Exception met de côté la même chaîne que l'original, mais dispose en plus de la possibilité de stocker dans l'exception la référence au fautif.

L'exemple suivant attrape la classe CustomException et met en utilisation ses informations sur MyClass :

```
public class Class1
{
  public void SomeFunction()
  {
    try
    {
      // . . . opérations préalables à la fonction exemple
      SomeOtherFunction();
      // . . . autres opérations. . .
    }
    catch(MyException me)
    {
      // vous avez toujours accès aux méthodes d'Exception
      string s = me.ToString();
      // mais vous avez aussi accès à toutes les propriétés et méthodes
      // de votre propre classe d'exceptions
      MyClass mo = me.MyCustomObject;
      // par exemple, demandez à l'objet MyClass de s'afficher lui-même
      string s = mo.GetDescription();
    }
  }
  public void SomeOtherFunction()
  {
    // création de myobject
    MyClass myobject = new MyClass();
    // . . . signale une erreur concernant myobject . . .
    throw new MyException("Erreur dans l'objet de MyClass", myobject);
    // . . . reste de la fonction . . .
  }
}
```

Dans ce fragment de code, SomeFunction() invoque SomeOtherFunction() de l'intérieur d'un bloc try. SomeOtherFunction() crée et utilise un objet myobject. Quelque part dans SomeOtherFunction(), une fonction de vérification d'erreur se prépare à envoyer une exception pour signaler qu'une condition d'erreur vient de se produire. Plutôt que de créer une simple Exception, SomeFunction() se sert de la toute nouvelle classe MyException, pour envoyer non seulement un message d'erreur, mais aussi l'objet myobject fautif.



Comme `Main()` est le point de départ du programme, il est bon de toujours en placer le contenu dans un bloc `try`. Toute exception qui ne sera pas "attrapée" ailleurs remontera finalement jusqu'à `Main()`. C'est donc votre dernière opportunité de récupérer une erreur avant qu'elle aboutisse à Windows, dont le message d'erreur sera beaucoup plus difficile à interpréter.

Le bloc `catch` situé à la fin de `Main()` attrape l'objet `Exception` et utilise sa méthode `ToString()` pour afficher sous forme d'une simple chaîne la majeure partie des informations sur l'erreur contenues dans l'objet `exception`.



La propriété `Exception.Message` retourne un sous-ensemble plus lisible, mais moins descriptif des informations sur l'erreur.

Cette version de la fonction `Factorial()` contient la même vérification pour un argument négatif que la précédente. Si l'argument est négatif, `Factorial()` met en forme un message d'erreur qui décrit le problème, incluant la valeur incriminée. `Factorial()` regroupe ensuite ces informations dans un objet `Exception` nouvellement créé, qu'elle envoie à la fonction appelante.

La sortie de ce programme apparaît comme suit (j'ai un peu arrangé les messages d'erreur pour les rendre plus lisibles) :

```
i = 6, factorielle = 720
i = 5, factorielle = 120
i = 4, factorielle = 24
i = 3, factorielle = 6
i = 2, factorielle = 2
i = 1, factorielle = 1
i = 0, factorielle = 0
Erreur fatale :
System.Exception: Argument négatif illicite passé à Factorial -1
   at Factorial(Int32 nValue) in c:\c#program\Factorial\class1.cs:line 23
   at FactorialException.Class1.Main(String[] args) in c:\c#program\Factorial\
   class1.cs:line 52
Appuyez sur Entrée pour terminer...
```

Les premières lignes affichent les véritables factorielles des nombres 6 à 0. La factorielle de -1 génère un message commençant par `Erreur fatale`, ce qui est susceptible d'attirer l'attention de l'utilisateur.

La première ligne du message d'erreur a été mise en forme dans la fonction `Factorial()` elle-même. Cette ligne décrit la nature du problème, en indiquant la valeur incriminée -1.

```
      throw new Exception("Description de l'erreur");
      // . . . suite de la fonction . . .
    }
  }
```

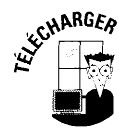
La fonction `SomeFunction()` contient un bloc de code identifié par le mot-clé `try`. Toute fonction appelée dans ce bloc, ou toute fonction qui l'appelle, est considérée comme faisant partie du bloc `try`.

Un bloc `try` est immédiatement suivi par le mot-clé `catch`, lequel est suivi par un bloc auquel le contrôle est passé si une erreur se produit en un endroit quelconque dans le bloc `try`. L'argument passé au bloc `catch` est un objet de la classe `Exception` ou d'une sous-classe de celle-ci.

À un endroit quelconque dans les profondeurs de `SomeOtherFunction()`, une erreur se produit. Toujours prête, la fonction signale une erreur à l'exécution en envoyant (`throw`) un objet `Exception` au premier bloc pour que celui-ci l'attrape (`catch`).

Puis-je avoir un exemple ?

Le programme `FactorialException` suivant met en évidence les éléments clés du mécanisme des exceptions :




```
// FactorialException - crée une fonction factorielle qui
//                       indique à Factorial() les arguments illicites
//                       en utilisant un objet Exception
using System;
namespace FactorialException
{
  // MyMathFunctions - collection de fonctions mathématiques
  //                   de ma création (pas encore grand-chose à montrer)
  public class MyMathFunctions
  {
    // Factorial - retourne la factorielle d'une valeur
    //             fournie
    public static double Factorial(int nValue)
    {
      // interdit les nombres négatifs
      if (nValue < 0)
      {
        // signale un argument négatif
        string s = String.Format(
             "Argument négatif illicite passé à Factorial {0}",
```

d'erreur que la fonction appelante ne teste pas. Bien sûr, en tant que programmeur en chef, je peux me laisser aller à proférer des menaces. Je me souviens d'avoir lu toutes sortes de livres de programmation regorgeant de menaces de bannissement du syndicat des programmeurs pour ceux qui ne s'occupent pas des codes d'erreur, mais tout bon programmeur FORTRAN sait bien qu'un langage ne peut obliger personne à vérifier quoi que ce soit, et que, très souvent, ces vérifications ne sont pas faites.

Souvent, même si je vérifie l'indication d'erreur retournée par `Factorial()` ou par toute autre fonction, la fonction appelante ne peut rien faire d'autre que de signaler l'erreur. Le problème est que la fonction appelante est obligée de tester toutes les erreurs possibles retournées par toutes les fonctions qu'elle appelle. Bien vite, le code commence à avoir cette allure là :

```
// appelle SomeFunction, lit l'erreur retournée, la traite
// et retourne
errRtn = someFunc();
if (errRtn == SF_ERROR1)
{
  Console.WriteLine("Erreur de type 1 sur appel à someFunc()");
  return MY_ERROR_1;
}
if (errRtn == SF_ERROR2)
{
  Console.WriteLine("Erreur de type 2 sur appel à someFunc()");
  return My_ERROR_2;
}
// appelle SomeOtherFunctions, lit l'erreur, retourne, et ainsi de suite
errRtn = someOtherFunc();
if (errRtn == SOF_ERROR1)
{
  Console.WriteLine("Erreur de type 1 sur appel à someFunc()");
  return MY_ERROR_3;
}
if (errRtn == SOF_ERROR2)
{
  Console.WriteLine("Erreur de type 1 sur appel à someFunc()");
  return MY_ERROR_4;
}
```

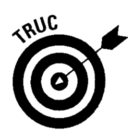
Ce mécanisme présente plusieurs inconvénients :

- Il est très répétitif.
- Il oblige le programmeur à inventer de nombreuses indications d'erreur et à en maîtriser l'emploi.

```
        if (dFactorial == MyMathFunctions.NON_INTEGER_VALUE)
        {
          Console.WriteLine
                      ("Factorial() a reçu un nombre non entier");
          break;
        }
        // affiche le résultat à chaque passage
        Console.WriteLine("i = {0}, factorielle = {1}",
                          i, MyMathFunctions.Factorial(i));
      }
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

`Factorial()` commence maintenant par effectuer une série de tests. Le premier regarde si la valeur passée est négative (0 est accepté parce qu'il donne un résultat raisonnable). Si oui, la fonction retourne immédiatement une indication d'erreur. Si non, la valeur de l'argument est comparée à sa version entière : si elles sont égales, c'est que la partie décimale de l'argument est nulle.

`Main()` teste le résultat retourné par `Factorial()`, à la recherche de l'indication éventuelle d'une erreur. Toutefois, des valeurs comme -1 et -2 n'ont guère de signification pour un programmeur qui effectue la maintenance de son code ou qui l'utilise. Pour rendre un peu plus parlante l'erreur retournée, la classe `MyMathFunctions` définit deux constantes entières. La constante `NEGATIVE_NUMBER` reçoit la valeur -1, et `NON_INTEGER_VALUE` reçoit la valeur -2. Cela ne change rien, mais l'utilisation des constantes rend le programme beaucoup plus lisible, en particulier la fonction appelante `Main()`.



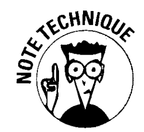
Dans la convention sur les noms Southern Naming Convention, les noms des constantes sont entièrement en majuscules, les mots étant séparés par un tiret de soulignement. Certains programmeurs, plus libéraux, refusent de faire allégeance, mais ce n'est pas la convention qui a des chances de changer.



Les constantes contenant les valeurs d'erreur sont accessibles par la classe, comme dans `MyMathClass.NEGATIVE_NUMBER`. Une variable de type `const` est automatiquement statique, ce qui en fait une propriété de classe partagée par tous les objets.

La fonction `Factorial()` signale maintenant qu'une valeur négative lui a été passée comme argument. Elle le signale à `Main()` qui se termine alors en affichant un message d'erreur beaucoup plus intelligible :

négatif. Ensuite, remarquez que les valeurs négatives ne croissent pas de la même manière que les valeurs positives. Manifestement, il y a quelque chose qui cloche.



Les résultats incorrects retournés ici sont assez subtils par rapport à ce qui aurait pu se produire. Si la boucle de `Factorial()` avait été écrite sous la forme `do {...} while (dValue != 0)`, le programme se serait planté en passant un nombre négatif. Bien sûr, je n'aurais jamais écrit une condition comme `while(dValue != 0)`, car les erreurs dues à l'approximation auraient pu faire échouer de toute façon la comparaison avec zéro.

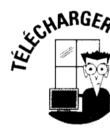
Retourner une indication d'erreur

Bien qu'elle soit assez simple, il manque à la fonction `Factorial()` une importante vérification d'erreur : la factorielle d'un nombre négatif n'est pas définie, pas plus que la factorielle d'un nombre non entier. La fonction `Factorial()` doit donc comporter un test pour vérifier que ces conditions sont remplies.

Mais que fera la fonction `Factorial()` avec une condition d'erreur si la chose se produit ? Elle connaîtra l'existence du problème, mais sans savoir comment il s'est produit. Le mieux que `Factorial()` puisse faire est de signaler les erreurs à la fonction qui l'appelle (peut-être celle-ci sait-elle d'où vient le problème).

La manière classique d'indiquer une erreur dans une fonction consiste à retourner une certaine valeur que la fonction ne peut pas autrement retourner. Par exemple, la valeur d'une factorielle ne peut pas être négative. La fonction `Factorial()` peut donc retourner -1 si un nombre négatif lui est passé, -2 pour un nombre non entier, et ainsi de suite. La fonction appelante peut alors examiner la valeur retournée : si cette valeur est négative, elle sait qu'une erreur s'est produite, et la valeur exacte indique la nature de l'erreur.

Le programme `FactorialErrorReturn` suivant contient les ajustements nécessaires :




```
// FactorialErrorReturn - crée une fonction factorielle qui
//                        retourne une indication d'erreur quand
//                        quelque chose ne va pas
using System;
namespace FactorialErrorReturn
{
  // MyMathFunctions - collection de fonctions mathématiques
  //                   de ma création (pas encore grand-chose à montrer)
```

Traiter une erreur à l'ancienne mode : la retourner

Ne pas signaler une erreur à l'exécution n'est jamais une bonne idée. Je dis bien *jamais* : si vous n'avez pas l'intention de déboguer vos programmes et si vous ne vous souciez pas qu'ils marchent, alors seulement c'est peut-être une bonne idée.

Le programme `FactorialError` suivant montre ce qui arrive quand les erreurs ne sont pas détectées. Ce programme calcule et affiche la fonction factorielle pour de nombreuses valeurs, dont certaines sont tout juste licites.



La factorielle du nombre N est égale à N * (N-1) * (N-2) * ... * 1. Par exemple, la factorielle de 4 est 4 * 3 * 2 * 1, soit 24. La fonction factorielle n'est valide que pour les nombres entiers naturels (positifs).




```
// FactorialWithError - créer et utiliser une fonction
//                       factorielle qui ne contient aucune
//                       vérification
using System;
namespace FactorialWithError
{
  // MyMathFunctions - collection de fonctions mathématiques
  //                    de ma création (pas encore grand-chose à montrer)
  public class MyMathFunctions
  {
    // Factorial - retourne la factorielle d'une valeur
    //             fournie
    public static double Factorial(double dValue)
    {
      // commence par donner la valeur 1 à un "accumulateur"
      double dFactorial = 1.0;
      // fait une boucle à partir de nValue en descendant de 1 chaque fois
      // pour multiplier l'accumulateur
      // par la valeur obtenue
      do
      {
        dFactorial *= dValue;
        dValue -= 1.0;
      } while(dValue > 1);
      // retourne la valeur stockée dans l'accumulateur
      return dFactorial;
    }
  }
  public class Class1
  {
    public static void Main(string[] args)
```

```
            components.Dispose();
          }
        }
        base.Dispose( disposing );
      }
      #region Windows Form Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Name = "Form1";
        this.Text = "Simple Editor";
      }
      #endregion
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
        Application.Run(new Form1());
      }
    }
  }
```

Je sais que le programme doit commencer par `static Main()`, qui se trouve ici tout en bas du listing. Voilà ce qui nourrit ma conviction que c'est ici qu'il faut commencer. La seule instruction que contient `Main()` crée un objet `Form1()` et le passe à une méthode `Application.Run()`. Je ne suis pas sûr de ce que fait `Run()`, mais je soupçonne fortement que la classe `Form1` correspond à la fenêtre Form1 que j'ai vue dans le Concepteur.



En fait, `Application.Run()` lance l'objet `Form` sur son propre thread d'exécution. Le thread initial s'arrête aussitôt que le nouveau `Form1` est créé. Le thread `Form1` se poursuit jusqu'à ce qu'il soit intentionnellement arrêté. Le programme `SimpleEditor` lui-même poursuit son exécution aussi longtemps que des threads définis par l'utilisateur sont actifs.

Le constructeur de `Form1` invoque une méthode `InitializeComponent()`. Tout code d'initialisation du programmeur doit être placé après cet appel (tout au moins, c'est ce que dit le commentaire).

Chapitre 20

Les dix plus importantes différences entre C# et C++

Dans ce chapitre :

Pas de données ni de fonctions globales.

Tous les objets sont alloués à partir du tas.

Les variables de type pointeur ne sont pas autorisées.

Vendez-moi quelques-unes de vos propriétés.

Je n'inclurai plus jamais un fichier.

Ne construisez pas, initialisez.

Définis soigneusement tes types de variable, mon enfant.

Pas d'héritage multiple.

Prévoir une bonne interface.

Le système des types unifiés.

Le langage C# est assez largement basé sur C++. Cela n'a rien d'étonnant, puisque Microsoft avait déjà fait Visual C++, qui a été le langage de programmation le plus répandu pour l'environnement Windows. Tous les meilleurs accros de la programmation s'en servaient. Mais ça fait déjà quelque temps que C++ avoue son âge.

C# n'est pas une couche de peinture sur une carcasse rouillée. Il comporte de nombreuses améliorations, à la fois par l'ajout de nouvelles fonctionnalités et par le remplacement de fonctionnalités déjà satisfaisantes par de meilleures. Voici les dix meilleures améliorations de C# par rapport à C++.



Un formulaire est une fenêtre contenant une barre de titre, et optionnellement des barres de défilement. Dans la terminologie de C#, une fenêtre n'est rien d'autre qu'un cadre rectangulaire dans lequel vous pouvez placer des images ou du texte. Une fenêtre n'a pas nécessairement des menus ou des étiquettes, ni même ces petits boutons Fermer, Réduire et Restaurer.

4. **Générez le programme que Windows vient de créer sur la base du modèle.**

   Vous pouvez me traiter de paranoïaque, mais je veux être certain que toutes les erreurs qui pourront apparaître par la suite seront réellement de mon fait et ne viendront pas de Visual Studio. Sans aucun doute, la solution va se générer sans encombre à ce stade. L'exécution de ce programme ne révèle rien d'autre qu'un formulaire vierge, doté de l'étiquette `Form1`. Il suffit de cliquer sur le bouton Fermer pour arrêter le programme.

   Le volet qui occupe la partie droite de l'affichage est la fenêtre Propriétés. Ça ne saute peut-être pas aux yeux, mais son contenu est en relation directe avec le formulaire qui est dans la partie gauche de l'affichage. Par exemple, vous pouvez voir que la propriété `Text` est `Form1`. Vous pouvez la modifier pour vous rendre compte de l'effet produit.

5. **Sélectionnez la propriété `Text`, et donnez-lui la valeur** Simple Editor.

   L'étiquette `Form1` contenue dans la barre de titre du formulaire devient Simple Editor.

6. **Générez à nouveau l'application, et exécutez-la.**

   Le nom du formulaire a changé, comme le montre la Figure 17.3.

SimpleEditor

Figure 17.3 : Changer la propriété `Text` du formulaire change le nom qui apparaît dans sa barre de titre.

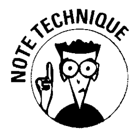
- Votre chien a mangé votre manuscrit. Plus simplement, vous avez oublié cette méthode ou vous n'en connaissiez pas l'existence. Soyez plus attentif la prochaine fois.
- Vous avez fait une faute de frappe dans le nom de la méthode ou vous lui avez passé de mauvais arguments.

Examinez l'exemple suivant :

```
Interface Me
{
  void aFunction(float);
}
public class MyClass : Me
{
  public void aFunction(double d)
  {
  }
}
```

La classe `MyClass` n'implémente pas la fonction `aFunction(float)` de l'interface. La fonction `aFunction(double)` ne compte pas, parce que les arguments ne correspondent pas.

Sur le métier remettez votre ouvrage, et passez en revue chacune de vos méthodes jusqu'à ce que toutes les méthodes de l'interface soient correctement implémentées.



Ne pas implémenter complètement une interface est essentiellement la même chose que d'essayer de créer une classe concrète à partir d'une classe abstraite sans redéfinir toutes les méthodes abstraites.

'methodName' : tous les chemins de code ne retournent pas nécessairement une valeur

Par ce message, C# vous dit que votre méthode a été déclarée non-void et que un ou plusieurs chemins d'exécution ne retournent rien. Cela peut se produire de l'une des deux manières suivantes :

- Vous avez une instruction `if` avec un `return` sans valeur spécifiée.
- Plus vraisemblablement, vous avez calculé une valeur et vous ne l'avez jamais retournée.

Le menu Édition a besoin des trois grandes options d'édition : Couper, Copier et Coller. D'autre part, tous les éditeurs comprennent les raccourcis clavier de ces trois options : Ctrl+X, Ctrl+C, et Ctrl+V, respectivement.

`SimpleEditor` aura également besoin d'un menu Format, comportant les options Gras et Italique pour mettre en forme le texte.

Fournir une aide véritable est une tâche difficile – beaucoup trop compliquée pour un éditeur simple comme `SimpleEditor`. Le menu d'aide de cette application devra se contenter du minimum absolu : l'option À propos de.

Dernière exigence : il nous faut un moyen de contrôler la taille de police. Voilà une chose qui laisse la place à un peu de fantaisie. En plus d'une simple fenêtre dans laquelle l'utilisateur peut entrer la taille de police souhaitée, `SimpleEditor` y ajoutera une sorte de barre munie d'un index que l'on peut faire glisser, que nous appellerons `TrackBar`. Pour obtenir 8 points, faites glisser l'index à l'extrémité gauche. Faites-le glisser à l'extrémité droite, et vous obtenez 24 points. (J'ai une autre raison de procéder ainsi : je veux vous montrer comment relier deux objets d'I/O de manière qu'un changement dans l'un soit répercuté dans l'autre.)

Ma solution

Avec les paramètres que j'ai décrits dans la section précédente, je suis arrivé à la solution montrée par la Figure 17.1. Vos propres résultats peuvent être différents selon vos goûts personnels.

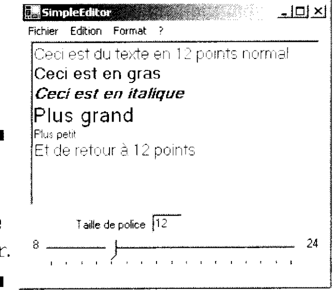


Figure 17.1 : Ma solution du problème `SimpleEditor`.

Dessiner la solution

Comme vous pouvez l'imaginer, j'ai dû passer par de nombreuses étapes pour arriver en partant de zéro à l'œuvre d'art montrée par la Figure 17.1.

cloche entre Visual Studio et le répertoire du programme. Fermez la solution, quittez Visual Studio, redémarrez, puis ouvrez à nouveau la solution. Si ça ne marche pas, je suis sincèrement désolé.

Le mot-clé new est requis sur 'subclassName. methodName', car il masque le membre hérité 'baseclassName.methodName'

Avec ce message, C# vous dit que vous avez surchargé une méthode dans une classe de base sans la redéfinir par une méthode qui la cache (voyez le Chapitre 13 pour en savoir plus à ce sujet). Regardez l'exemple suivant :

```
public class BaseClass
{
  public void Function()
  {
  }
}
public class SubClass : BaseClass
{
  public void Function()
  {
  }
}
public class MyClass
{
  public void Test()
  {
    SubClass sb = new SubClass();
    sb.Function();
  }
}
```

La fonction `Test()` ne peut pas accéder à la méthode `BaseClass. Function()` à partir de l'objet `sb` d'une sous-classe, car elle est redéfinie par `SubClass. Function()`. Vous aviez l'intention de faire l'une des choses suivantes :

- Vous vouliez redéfinir la méthode de la classe de base. Dans ce cas, ajoutez le mot-clé `new` dans la définition de `SubClass`, comme dans l'exemple suivant :

```
public class SubClass : BaseClass
{
```

Quel est le problème ?

Il m'a fallu une longue et difficile réflexion (au moins un quart d'heure) pour imaginer un problème qui mette en lumière la puissance de C# sans me faire prendre du poids. Le voici : créer un éditeur simple que nous appellerons `SimpleEditor`. Il aura les caractéristiques suivantes :

- L'utilisateur peut entrer et effacer du texte (sinon, ce ne serait pas vraiment un éditeur).
- L'utilisateur peut couper et coller du texte, non seulement dans `SimpleEditor`, mais aussi entre `SimpleEditor` et d'autres applications, par exemple Word.
- `SimpleEditor` supporte les polices en gras, en italique ou les deux.
- L'utilisateur peut sélectionner une taille de police de 8 à 24 points. Ces limites sont arbitraires, mais il s'agit ici de ne pas aller trop loin en nombre de points.
- `SimpleEditor` ne doit pas vous permettre de quitter sans vous avoir demandé poliment d'enregistrer le fichier que vous venez de modifier (mais vous restez libre de quitter sans enregistrer si c'est bien ce que vous voulez).

Exposer le problème

Chaque fois que vous êtes devant un problème à résoudre, vous devez commencer par vous mettre devant le tableau noir et réfléchir sérieusement aux obstacles à franchir. Dans le cas d'une application Windows, cette tâche se divise en trois étapes :

1. **Décrivez le problème en détail.**

   Ces détails sont les spécifications auxquelles doit se conformer l'application. Au cours de la programmation, vous pourrez être tenté d'ajouter une fonctionnalité ici ou là. Résistez. Cette maladie s'appelle *fonctionnalite*. Tout en avançant, notez les améliorations possibles pour une version future, mais l'ajout de fonctionnalités en cours de route fait courir le risque de créer une application qui finit par être tout à fait autre chose que ce qu'elle était censée être au départ.

Par défaut, un membre d'une classe est `private`, et une classe est `internal`. Aussi, `nPrivateMember` est toujours privé dans l'exemple suivant :

```
class MyClass
{
  public void SomeFunction()
  {
    YourClass uc = new YourClass();
    // ceci ne fonctionne pas correctement parce que MyClass
    // ne peut pas accéder au membre privé
    uc.nPrivateMember = 1;
  }
}
public class YourClass
{
  int nPrivateMember = 0;    // ce membre est toujours privé
}
```

En outre, même si `SomeFunction()` est déclarée `public`, on ne peut pas y accéder à partir d'une classe d'un autre module, car `MyClass` elle-même est interne.

La morale de l'histoire est la suivante : "Spécifiez toujours le niveau de protection de vos classes et de leurs membres." Et nous avons un lemme qui dit : "Ne déclarez pas de membres publics dans une classe qui elle-même est interne. Ça n'apporte que de la confusion."

Utilisation d'une variable locale non assignée 'n'

Comme il le dit si clairement, ce message indique que vous avez déclaré une variable mais que vous ne lui avez pas donné de valeur initiale. C'est en général un simple oubli, mais ça peut aussi se produire lorsque vous voulez vraiment passer une variable comme argument `out` à une fonction :

```
public class MyClass
{
  public void SomeFunction()
  {
    int n;
    // ceci fonctionne parce que C# ne retourne une valeur que dans n
    // il ne passe pas une valeur dans la fonction
    SomeOtherFunction(out n);
  }
  public void SomeOtherFunction(out int n)
  {
```

Dans cette partie...

Comprendre C# est une chose, apprendre à écrire une application Windows complète avec tous ses assemblages et ses décorations bien en place en est une autre. Rien que pour le plaisir, la cinquième partie vous guide pas à pas dans l'utilisation de C# avec l'interface Visual Studio afin de créer une application Windows "qui ne soit pas triviale". Vous serez fier du résultat, même si vos enfants n'appellent pas leurs copains pour le voir.

```
      // ceci fonctionne très bien
      float fResult = 2 * f;
      return fResult;
    }
  }
```

La constante 2 est de type int. Un int multiplié par un float donne un float, qui peut être stocké dans la variable fResult, de type float.

Le message d'erreur de conversion implicite peut aussi apparaître lorsque vous effectuez des opérations sur des types "non naturels". Par exemple, vous ne pouvez pas additionner deux variables de type char, mais C# peut convertir pour vous une variable de type char en une valeur de type int lorsque c'est nécessaire pour réaliser l'opération. Ce qui conduit à ceci :

```
class MyClass
{
  static public void SomeFunction()
  {
    char c1 = 'a';
    char c2 = 'b';
    // je ne sais même pas ce que ceci pourrait vouloir dire ; c'est illicite
    // mais pas pour la raison que vous croyez
    char c3 = c1 + c2;
  }
}
```

Additionner deux caractères n'a en soi aucun sens, mais C# essaie quand même. Comme l'addition n'est pas définie pour le type char, il convertit c1 et c2 en valeurs int avec lesquelles il effectue l'addition. Malheureusement, la valeur int qui en résulte ne peut pas être convertie à nouveau en type char sans intervention extérieure.

La plupart des conversions, mais pas toutes, se passent très bien avec un cast explicite. La fonction suivante fonctionne sans se plaindre :

```
class MyClass
{
  static public float FloatTimes2(float f)
  {
    // ceci fonctionne très bien avec le cast explicite
    float fResult = (float)(2.0 * f);
    return fResult;
  }
}
```

```
    static public void MyFunction(Student s)
    {
      Console.WriteLine("Nom de l'étudiant = " + s.sStudentName);
      Console.WriteLine("Numéro d'identification de l'étudiant = " + s.nId);
    }
  }
```

Le problème est ici que `MyFunction()` fait référence à un membre donnée `nId` au lieu du véritable membre donnée `nID`. Vous voyez la ressemblance, mais C# ne la voit pas. Le programmeur a écrit `nId`, mais il n'y a pas de `nId`, et puis c'est tout.

Un peu moins populaire, mais également dans le Top 10, vous avez aussi la possibilité que la variable ait été déclarée dans une portée différente :

```
class MyClass
{
  static public void AverageInput()
  {
    int nCount = 0;
    while(true)
    {
      // lit un nombre
      string s = Console.ReadLine();
      int n = Int32.Parse(s);
      // quitte si l'utilisateur entre un nombre négatif
      if (n < 0)
      {
        break;
      }
      // ajoute la valeur entrée
      nSum += n;
      nCount++;
    }
    // affiche maintenant les résultats
    Console.WriteLine("Le total est " + nSum);
    Console.WriteLine("La moyenne est " + nSum / nCount);
    // ceci produit un message d'erreur de génération
    Console.WriteLine("La valeur finale était " + s);
  }
}
```

La dernière ligne de cette fonction est incorrecte. Le problème est qu'une variable est limitée à la portée dans laquelle elle est définie. La variable `s` n'est pas définie en dehors de la boucle `while()`.

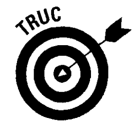
que le programme donne la sortie attendue. Une fois que le programme a lu le fichier, il se termine. Si l'utilisateur veut lire un autre fichier, il lui suffit d'exécuter à nouveau le programme.

Le programme commence par une boucle `while`, comme son cousin `FileWrite`. Dans cette boucle, il va chercher le nom de fichier entré par l'utilisateur. Si le nom de fichier est vide, le programme envoie un message d'erreur : `Vous avez entré un nom de fichier vide`. Dans le cas contraire, le nom de fichier est utilisé pour ouvrir un objet `FileStream` en mode de lecture. L'appel `File.Open()` est ici le même que celui utilisé dans `FileWrite` :

- Le premier argument est le nom du fichier.
- Le deuxième argument est le modèle du fichier. Le mode `FileMode.Open` dit : "Ouvrir le fichier s'il existe, sinon envoyer une exception." L'autre possibilité est `OpenNew`, qui crée un fichier de longueur nulle si celui-ci n'existe pas déjà. Personnellement, je n'ai jamais rencontré le besoin de ce mode (qui veut lire un fichier vide ?), mais chacun mène sa barque comme il l'entend.
- Le dernier argument indique que je veux lire à partir de ce `FileStream`. Les autres solutions sont `Write` et `ReadWrite`.

L'objet `FileStream fs` résultant est alors inséré dans un objet `StreamReader sr` qui offre des méthodes pratiques pour accéder au fichier texte.

Toute cette section d'ouverture de fichier est enchâssée dans un bloc `try`, lui-même enchâssé dans une boucle `while`, insérée dans une énigme. Ce bloc `try` est strictement réservé à l'ouverture de fichier. Si une erreur se produit pendant le processus d'ouverture, l'exception est attrapée, un message d'erreur est affiché, et le programme reprend au début de la boucle pour demander à nouveau un nom de fichier à l'utilisateur. Toutefois, si le processus aboutit à un objet nouveau-né `StreamReader` en bonne santé, la commande `break` fait sortir de la logique d'ouverture de fichier et fait passer le chemin d'exécution du programme à la section de lecture.



`FileRead` et `FileWrite` représentent deux manières différentes de traiter des exceptions de fichier. Vous pouvez insérer tout le programme de traitement de fichier dans un même bloc `try`, comme dans `FileWrite`, ou bien vous pouvez donner son propre bloc `try` à la section d'ouverture de fichier. Cette dernière solution est généralement la plus facile, et elle permet de générer un message d'erreur plus précis.

Une fois le processus d'ouverture de fichier terminé, le programme `FileRead` lit une ligne de texte dans le fichier en utilisant l'appel

Chapitre 19

Les dix erreurs de génération les plus courantes (et comment y remédier)

Dans ce chapitre :

'className' ne contient pas de définition pour 'memberName'.

Impossible de convertir implicitement le type 'x' en 'y'.

'className.memberName' est inaccessible en raison de son niveau de protection.

Utilisation d'une variable locale non assignée 'n'.

Le fichier 'programName.exe' ne peut pas être copié dans le répertoire d'exécution. Le processus ne peut pas...

Le mot-clé new est requis sur 'subclassName.methodName', car il masque le membre hérité 'baseclassName.methodName'.

'subclassName' : ne peut pas hériter de la classe scellée 'baseclassName'.

'className' n'implémente pas le membre d'interface 'methodName'.

'methodName' : tous les chemins de code ne retournent pas nécessairement une valeur.

} attendue.

De façon très scolaire, C# fait de son mieux pour trouver des erreurs dans votre code. Il se jette sur les fautes de syntaxe comme un fauve sur sa proie. En dehors des erreurs vraiment bêtes, comme essayer de compiler votre liste de commissions, on a l'impression d'entendre toujours le même cri de protestation, inlassablement.

Ce chapitre présente dix messages d'erreur de génération que l'on rencontre souvent, mais quelques avertissements s'imposent. Tout d'abord, C# est

Améliorez votre compréhension et votre vitesse de lecture avec StreamReader

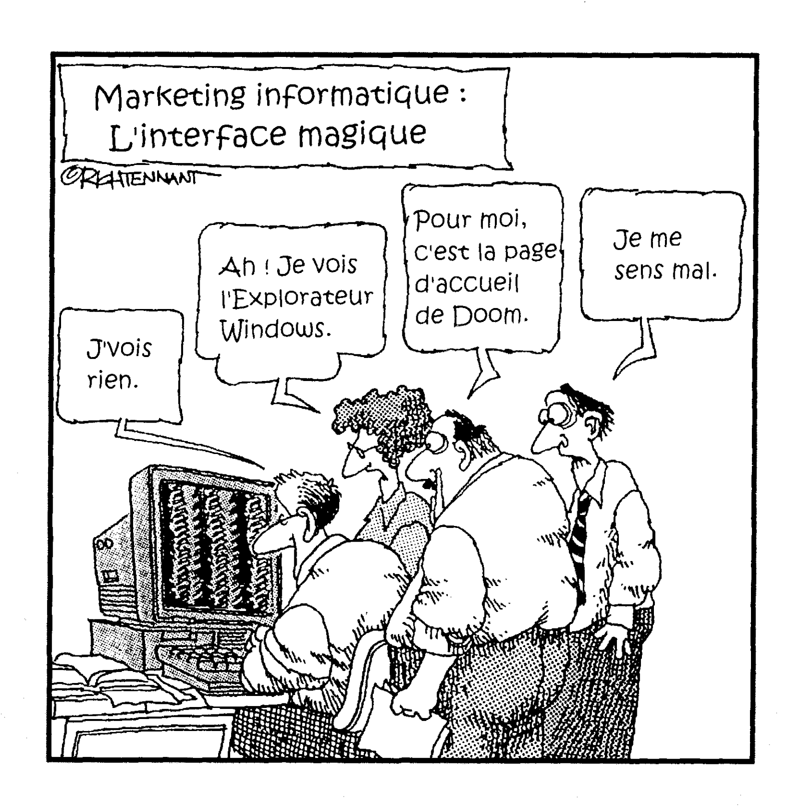
Il est très agréable d'écrire sur un fichier, mais c'est plutôt inutile si vous ne pouvez pas lire le fichier par la suite. Le programme FileRead suivant affiche sur la console ce qu'il lit dans le fichier. Ce programme lit un fichier texte comme celui que crée FileWrite :



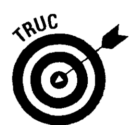

```
// FileRead - lit un fichier texte et l'écrit
//            sur la console
using System;
using System.IO;
namespace FileRead
{
  public class Class1
  {
    public static void Main(string[] args)
    {
      // il nous faut un objet pour lire le fichier
      StreamReader sr;
      string sFileName = "";
      // continue à essayer de lire un nom de fichier jusqu'à ce qu'il en
      // trouve un (la seule manière de quitter pour l'utilisateur est
      // d'arrêter le programme en appuyant sur Ctrl + C)
      while(true)
      {
        try
        {
          // lit le nom du fichier d'entrée
          Console.Write("Entrez le nom d'un fichier texte à lire :");
          sFileName = Console.ReadLine();
          // l'utilisateur n'a rien entré ; envoie une erreur
          // pour lui dire que ce n'est pas satisfaisant
          if (sFileName.Length == 0)
          {
            throw new IOException("Vous avez entré un nom de fichier vide");
          }
          // ouvre un flux de fichier pour la lecture ; ne crée pas
          // le fichier s'il n'existe pas déjà
          FileStream fs = File.Open(sFileName,
            FileMode.Open,
            FileAccess.Read);
          // convertit ceci en StreamReader - ce sont les trois premiers
          // octets du fichier qui seront utilisés pour indiquer
          // l'encodage utilisé (mais pas le langage)
          sr = new StreamReader(fs, true);
```

Sixième partie

Petits suppléments par paquets de dix

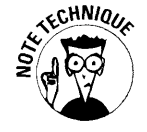


- **Le type d'accès :** Un fichier peut être ouvert pour la lecture, l'écriture ou les deux.



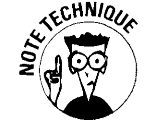
`FileStream` dispose de nombreux constructeurs, dont chacun correspond par défaut à un ou deux des arguments de mode et d'accès. Toutefois, à mon humble avis, il vaut mieux spécifier explicitement ces arguments, car ils ont un effet important sur le programme.

Dans la ligne suivante, le programme insère dans un objet `StreamWriter`, `sw`, l'objet `FileStream` qu'il vient d'ouvrir. La classe `StreamWriter` permet d'insérer les objets `FileStream`, afin de fournir un ensemble de méthodes pour traiter du texte. Le premier argument du constructeur `StreamWriter` est l'objet `FileStream`. Le deuxième spécifie le type d'encodage à utiliser. L'encodage par défaut est UTF8.



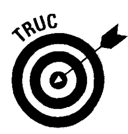
Il n'est pas nécessaire de spécifier l'encodage pour lire un fichier. `StreamWriter` inscrit le type d'encodage dans les trois premiers octets du fichier. À l'ouverture du fichier, ces trois octets sont lus pour déterminer l'encodage.

Le programme `FileWrite` commence alors à lire sous forme de chaînes les lignes saisies sur la console. Le programme arrête de lire lorsque l'utilisateur entre une ligne blanche, mais jusque-là il continue à absorber tout ce qu'on lui donne pour le déverser dans l'objet `StreamWriter sw` en utilisant la méthode `WriteLine()`.



La similitude entre `StreamWriter.WriteLine()` et `Console.WriteLine()` n'est pas qu'une coïncidence.

Enfin, le fichier est fermé par l'instruction `sw.Close()`.



Remarquez que le programme donne à la référence `sw` la valeur `null` à la fermeture du fichier. Un objet fichier est parfaitement inutile une fois que celui-ci a été fermé. Il est de bonne pratique de donner à la référence la valeur `null` une fois qu'elle est devenue invalide, afin de ne pas essayer de l'utiliser à nouveau dans l'avenir.

Le bloc `catch` qui suit la fermeture du fichier est un peu comme un gardien de but : il est là pour attraper toute erreur de fichier qui aurait pu se produire en un endroit quelconque du programme. Ce bloc émet un message d'erreur, contenant le nom du fichier qui en est responsable. Mais il ne se contente pas d'indiquer simplement le nom du fichier : il vous donne son chemin d'accès complet, en ajoutant à l'aide de la méthode `Path.Combine()` le nom du répertoire courant avant le nom de

```
private void ApplicationWindowClosing(object sender,
System.ComponentModel.CancelEventArgs e)
{
  if (!IsChangeOK())
  {
    e.Cancel = true;
  }
}
```

5. **Générez le programme et exécutez-le.**

6. **Entrez du texte et cliquez sur le bouton de fermeture de la fenêtre.**

   Vous voyez apparaître le même message d'avertissement que celui produit par la commande Fichier/Quitter.

Réaliser vos propres applications Windows

La création d'un programme comme `SimpleEditor` nécessite de nombreuses étapes, et c'est une application relativement simple. Or, il est en fait beaucoup plus facile de créer une application Windows avec Visual Studio .NET qu'avec les outils que nous connaissions auparavant. Il y a quatre ou cinq ans, même une petite chose comme l'affichage d'une simple boîte de message était difficile. Au temps anciens de Windows 3.1, c'était une montagne.

Quoi qu'il en soit, ne vous laissez pas décourager. Pensez à la présentation et au fonctionnement que vous attendez de votre application. Mettez tout cela par écrit. Alors seulement, vous pouvez utiliser le Concepteur de formulaires pour dessiner les éléments qui la composent. Utilisez la fenêtre Propriétés pour identifier les propriétés, statiques et dynamiques, que vous voulez définir afin que l'application fonctionne exactement comme vous voulez.

```
            //                  FileAccess.Write,
            //                  FileAccess.ReadWrite
            FileStream fs = File.Open(sFileName,
                                 FileMode.CreateNew,
                                 FileAccess.Write);
            // génère un flux de fichier avec des caractères UTF8
            sw = new StreamWriter(fs, System.Text.Encoding.UTF8);
            // lit une chaîne à la fois, et envoie chacune au
            // FileStream ouvert pour écriture
            Console.WriteLine("Entrez du texte ; ligne blanche pour arrêter");
            while(true)
            {
              // lit la ligne suivante sur la console ;
              // quitte si la ligne est blanche
              string sInput = Console.ReadLine();
              if (sInput.Length == 0)
              {
                break;
              }
              // écrit sur le fichier de sortie la ligne qui vient d'être lue
              sw.WriteLine(sInput);
            }
            // ferme le fichier que nous avons créé
            sw.Close();
            sw = null;
          }
          catch(IOException fe)
          {
            // une erreur s'est produite quelque part pendant
            // le traitement du fichier - indique à l'utilisateur
            // le nom complet du fichier :
            // ajoute au nom du répertoire par défaut
            // celui du fichier
            string sDir = Directory.GetCurrentDirectory();
            string s = Path.Combine(sDir, sFileName);
            Console.WriteLine("Erreur sur le fichier{0}", s);
            // affiche maintenant le message d'erreur de l'exception
            Console.WriteLine(fe.Message);
          }
        }
        // attend confirmation de l'utilisateur
        Console.WriteLine("Appuyez sur Entrée pour terminer...");
        Console.Read();
      }
    }
  }
```

`FileWrite` utilise l'espace de nom `System.IO` ainsi que `System`. `System.IO` contient les fonctions d'I/O sur les fichiers.

contenu de la zone de texte. Dans le Concepteur de formulaires, sélectionnez encore une fois le composant RichTextBox, puis, dans la fenêtre Propriétés, attribuez le nom de méthode TextChanged à la propriété TextChanged. Cette méthode ne fait rien de plus que d'assigner la valeur voulue à notre indicateur :

```
// cette méthode est appelée lorsque le texte est modifié
private void TextChanged(object sender, System.EventArgs e)
{
  bTextChanged = true;
}
```

Il est temps d'essayer le résultat :

1. **Dans le programme préalablement régénéré, ouvrez un fichier RTF.**

2. **Faites une modification quelconque dans la zone de texte.**

3. **Sélectionnez Fichier/Quitter.**

   La boîte de dialogue d'avertissement apparaît, comme dans la Figure 18.8. Vous pouvez pousser un soupir de soulagement.

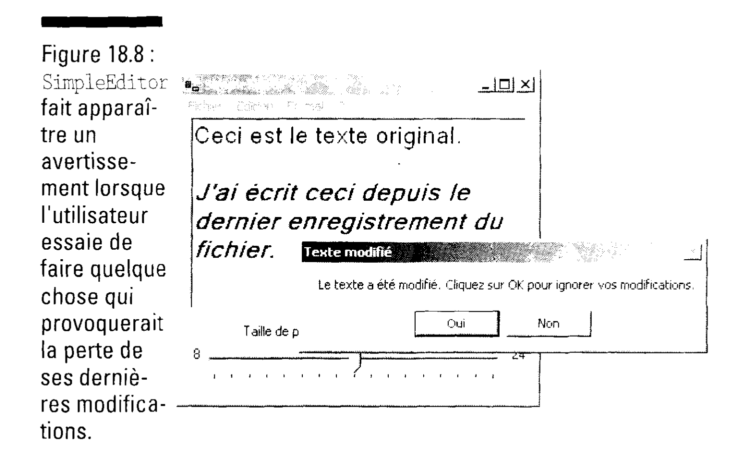
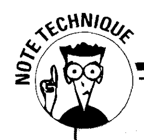


Figure 18.8 : SimpleEditor fait apparaître un avertissement lorsque l'utilisateur essaie de faire quelque chose qui provoquerait la perte de ses dernières modifications.

4. **Cliquez sur Oui, et le programme se ferme.**

5. **Répétez le processus en commençant par l'étape 1. Cliquez sur Non, et le programme continue comme si rien ne s'était passé.**



I/O asynchrones : est-ce que ça vaut la peine d'attendre ?

Normalement, un programme attend qu'une requête d'I/O sur un fichier soit satisfaite avant de poursuivre son exécution. Appelez une méthode `read()`, et vous ne récupérerez généralement pas le contrôle aussi longtemps que les données du fichier ne seront pas installées à bord en sécurité. C'est ce que l'on appelle une *I/O synchrone*.

Avec C#, les classes de `System.IO` supportent également les I/O asynchrones. En les utilisant, l'appel à `read()` restitue immédiatement le contrôle pour permettre au programme de poursuivre son exécution pendant que la requête d'I/O est satisfaite à l'arrière-plan. Le programme est libre de vérifier l'état d'un indicateur pour savoir si la requête d'I/O a abouti.

C'est un peu comme de faire cuire un hamburger. Avec des I/O synchrones vous mettez la viande hachée à cuire sur la plaque chauffante, vous la surveillez jusqu'à ce qu'elle soit cuite, et c'est seulement à partir de là que vous pouvez vous mettre à couper les oignons qui vont aller dessus.

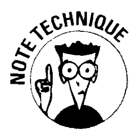
Avec des I/O asynchrones, vous pouvez couper les oignons pendant que la viande hachée est en train de cuire. De temps en temps, vous jetez un coup d'œil pour voir si elle est cuite. Le moment venu, vous abandonnez un instant vos oignons, et vous prenez la viande sur la plaque chauffante pour la mettre sur le pain.

Les I/O asynchrones peuvent améliorer significativement les performances d'un programme, mais elles ajoutent un niveau supplémentaire de complexité.

Utiliser `StreamWriter`

Les programmes génèrent deux sortes de sortie. Certains programmes écrivent des blocs de données dans un pur format binaire. Ce type de sortie est utile pour stocker des objets d'une manière efficace.

Beaucoup de programmes, sinon la plupart, lisent et écrivent des chaînes de texte, lisibles par un être humain. Les classes de flux `StreamWriter` et `StreamReader` sont les plus souples des classes accueillantes pour l'homme.



Les données lisibles par un être humain étaient antérieurement des chaînes ASCII, ou, un peu plus tard, ANSI. Ces deux sigles se réfèrent aux organisations de standardisation qui ont défini ces formats. Toutefois, le codage ANSI ne permet pas d'intégrer les alphabets venant de plus loin que l'Autriche à l'Est, et de plus loin que Hawaï à l'Ouest. Il ne peut contenir que l'alphabet latin. Il ne dispose pas de l'alphabet cyrillique, hébreu,

à cet indicateur lorsqu'un fichier est lu (rien n'y a encore été modifié). Saisir du texte, couper du texte ou coller du texte dans la zone de texte assigne `true` à cet indicateur. L'exécution de la commande Enregistrer assigne à nouveau `false` à l'indicateur. Donnons à celui-ci le nom `bTextChanged`.

Naturellement, l'utilisateur peut toujours quitter le programme, même si la dernière version de son travail n'a pas été enregistrée, mais il devra maintenant décider consciemment de le faire (autrement dit, cliquer sur un bouton OK dans une fenêtre d'avertissement).

C'est ce que fait avec simplicité la méthode `IsChangeOK()` suivante :

```
// la méthode suivante garantit que l'utilisateur ne perdra pas
// ses modifications par inadvertance, en affichant un message
// si la RichTextBox n'est pas "propre"
bool bTextChanged = false;
private bool IsChangeOK()
{
  // il est toujours sans inconvénient de quitter le programme
  // si rien n'a été modifié
  if (bTextChanged == false)
  {
    return true;
  }
  // mais quelque chose a été modifié ; le programme demande
  // à l'utilisateur ce qu'il veut en faire
  DialogResult dr = MessageBox.Show("Le texte a été modifié. "
                        + "Cliquez sur OK pour ignorer vos modifications.",
                          "Texte modifié",
                          MessageBoxButtons.YesNo);
  return dr == DialogResult.Yes;
}
```

`IsChangeOK()` retourne `true` si l'utilisateur est d'accord pour perdre les dernières modifications du contenu de `RichTextBox`. Mais avant tout, si l'indicateur de modification a pour valeur `false`, c'est que rien n'a changé depuis la dernière commande Fichier/Enregistrer, et que rien ne peut donc être perdu.

Si quelque chose a été modifié, la fonction ouvre une boîte de dialogue `MessageBox` pour demander à l'utilisateur ce qu'il veut faire : continuer et perdre ses modifications, ou annuler et les conserver. La classe `MessageBox` est aussi simple que cette boîte de dialogue. La méthode `Show()` ouvre une boîte de dialogue avec le titre et le message spécifiés. La propriété `YesNo` dit : "Faire cette boîte de dialogue avec un bouton Oui et un bouton Non." Si

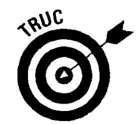
- Une méthode déclarée internal est accessible par toutes les classes du même espace de nom. Aussi, l'appel class2.D_internal() n'est pas autorisé. L'appel class3.C_internal() est autorisé parce que Class3 fait partie de l'espace de nom AccessControl.

- Le mot-clé internal protected combine l'accès internal et l'accès protected. Aussi, l'appel class1.E_internalprotected() est autorisé, parce que Class1 étend Class2 (c'est la partie protected). L'appel class3.E_internalprotected() est également autorisé, parce que Class1 et Class3 font partie du même espace de nom (c'est la partie internal).

- La déclaration de Class3 comme internal a pour effet de réduire l'accès à celle-ci à internal, ou moins. Aussi, les méthodes public deviennent internal, alors que les méthodes protected deviennent internal protected.

Ce programme donne la sortie suivante :

```
Class2.A_public
Class2.B_protected
Class1.C_private
Class3.D_internal
Class2.E_internalprotected
Class3.E_internalprotected
Appuyez sur Entrée pour terminer...
```

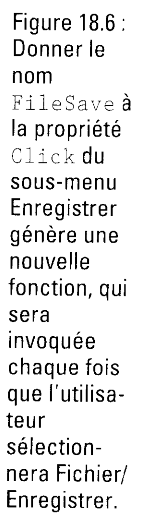


Déclarez toujours les méthodes avec un accès aussi restreint que possible. Une méthode privée peut être modifiée à volonté sans inquiéter de l'effet que cela pourrait avoir sur d'autres classes. Une classe ou une méthode interne de MathRoutines est utilisé par d'autres classes de nature mathématique. Si vous n'êtes pas convaincu de la sagesse du couplage faible entre les classes, allez voir le Chapitre 11.

Rassembler des données dans des fichiers

Les applications console de ce livre reçoivent essentiellement leurs entrées de la console, et y envoient de même leur sortie. Les programmes des autres sections que celle-ci ont mieux à faire (ou autre chose) que de vous embêter avec des manipulations de fichiers. Je ne veux pas les obscurcir avec la question supplémentaire des entrées/sorties (I/O). Toutefois, les applications console qui n'effectuent pas d'opération d'entrée/sortie sur des fichiers sont à peu près aussi courantes que les phoques dans la Seine.

4. **Faites la même chose pour l'option de menu Fichier/Ouvrir, en utilisant le nom de fonction** FileOpen.



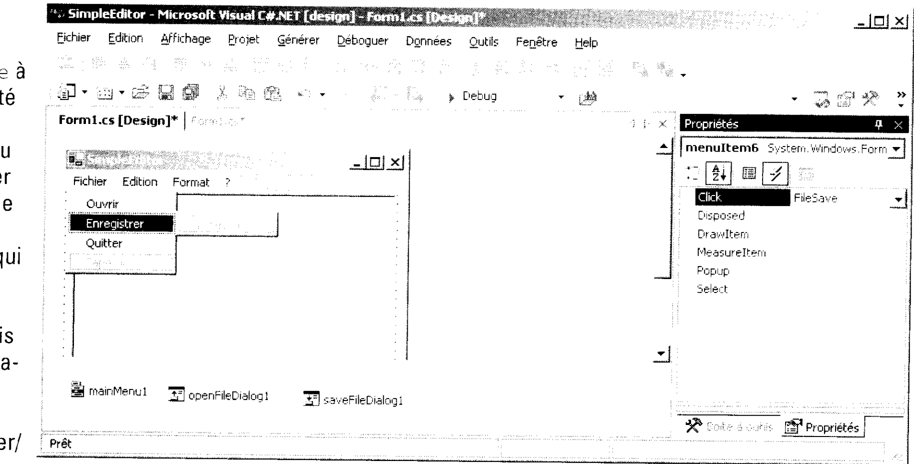


Figure 18.6 : Donner le nom `FileSave` à la propriété `Click` du sous-menu Enregistrer génère une nouvelle fonction, qui sera invoquée chaque fois que l'utilisateur sélectionnera Fichier/Enregistrer.

5. **Implémentez les méthodes** `FileOpen()` **et** `FileSave()` **comme suit :**

```
private void FileOpen(object sender, System.EventArgs e)
{
  OpenAndReadFile();
}

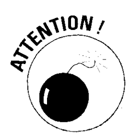
private void FileSave(object sender, System.EventArgs e)
{
  SaveSpecifiedFile();
}
```

Ce sont ces deux fonctions simples qui permettent à `SimpleEditor` d'être un véritable éditeur : il peut maintenant lire et écrire des fichiers. Par exemple, bien que ça ne se voie pas, le texte que montre la Figure 18.7 a en fait été écrit dans Word (enregistré au format RTF, bien sûr), et lu en utilisant la commande Fichier/Ouvrir.

```
      // la même classe
      //class2.C_private();
      class1.C_private();
      // les méthodes internes ne sont accessibles que par
      // les classes du même espace de nom
      //class2.D_internal();
      class3.D_internal();
      // les méthodes internes protégées sont accessibles
      // soit par la hiérarchie d'héritage soit par
      // toute classe du même espace de nom
      class1.E_internalprotected();
      class3.E_internalprotected();
      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
      return 0;
    }
    public void C_private()
    {
      Console.WriteLine("Class1.C_private");
    }
  }
  // Class3 - une classe interne est accessible aux autres
  //          classes du même espace de nom, mais
  //          pas aux classes externes qui utilisent cet
  //          espace de nom
  internal class Class3
  {
    // la déclaration d'une classe comme interne force toutes
    // les méthodes publiques à être également internes
    public void A_public()
    {
      Console.WriteLine("Class3.A_public");
    }
    protected void B_protected()
    {
      Console.WriteLine("Class3.B_protected");
    }
    internal void D_internal()
    {
      Console.WriteLine("Class3.D_internal");
    }
    public void E_internalprotected()
    {
      Console.WriteLine("Class3.E_internalprotected");
    }
  }
}
```

```
    {
      // affiche le message d'erreur dans la fenêtre de texte elle-même
      richTextBox1.Text = "Impossible de lire le fichier\n";
      richTextBox1.Text += e.Message;
      bReturnValue = true;
    }
    return bReturnValue;
  }
```

Cette fonction commence par afficher une boîte de dialogue OpenFileDialog. Si la boîte de dialogue retourne OK, la fonction essaie d'ouvrir le fichier sélectionné en utilisant la méthode OpenFile(). Si cette méthode retourne un objet Stream valide, OpenAndReadFile() insère l'objet Stream dans un StreamReader, plus pratique. Elle lit ensuite tout le contenu du fichier, puis elle le copie dans RichTextBox en l'assignant à sa propriété Rtf. Enfin, OpenAndReadFile() ferme le fichier.



Un éditeur qui lit tout le fichier en mémoire est beaucoup plus facile à écrire qu'un éditeur qui en laisse la majeure partie sur le disque. En tout cas, C# ne limite pas son composant RichTextBox au tampon mémoire si frustrant de 64 Ko du Bloc-notes.

Si une erreur d'I/O de fichier se produit, OpenAndReadFile() envoie le message erreur dans la zone de texte RichTextBox elle-même.



Nous avons ajouté le composant OpenFileDialog à SimpleEditor en le faisant glisser depuis la Boîte à outils.

Écrire un fichier RTF

La classe SaveFileDialog offre des méthodes qui sont tout aussi pratiques que celles qui servent à ouvrir les fichiers :

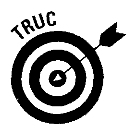
```
// enregistre le fichier sur le disque ; retourne true en cas de succès
// (cette fois, ne pas essayer d'attraper une exception - je ne
// saurais de toute façon pas quoi en faire)
private bool SaveSpecifiedFile()
{
  bool bReturnValue = false;
  // ce code est construit sur le même modèle que OpenAndReadFile()
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    System.IO.Stream strOutput = saveFileDialog1.OpenFile();
```

Utiliser un espace de nom avec le mot-clé using

Se référer à une classe par son nom pleinement qualifié peut devenir un peu fastidieux. Le mot-clé using de C# vous permet d'éviter ce pensum. La commande using ajoute l'espace de nom spécifié à une liste d'espaces de nom par défaut que C# consulte pour essayer de résoudre un nom de classe. L'exemple de programme suivant se compile sans une plainte :

```
namespace Paint
{
  public class PaintColor
  {
    public PaintColor(int nRed, int nGreen, int nBlue) {}
    public void Paint() {}
    public static void StaticPaint() {}
  }
}
namespace MathRoutines
{
  // ajoute Paint aux espaces de nom dans lesquels on cherche
  // automatiquement
  using Paint;
  public class Test
  {
    static public void Main(string[] args)
    {
      // crée un objet dans un autre espace de nom - il n'est
      // pas nécessaire de faire figurer le nom de l'espace de nom, car
      // celui-ci est inclus dans une instruction "using"
      PaintColor black = new PaintColor(0, 0, 0);
      black.Paint();
      PaintColor.StaticPaint();
    }
  }
}
```

La commande using dit : "Si vous ne trouvez pas la classe spécifiée dans l'espace de nom courant, voyez si vous la trouvez dans celui-ci." Vous pouvez spécifier autant d'espaces de nom que vous voulez, mais toutes les commandes using doivent apparaître l'une après l'autre tout à fait au début du programme.



Tous les programmes commencent par la commande using System;. Elle donne au programme un accès automatique à toutes les fonctions de la bibliothèque système, comme WriteLine().

appelle des *filtres*. La dernière option de la liste des filtres est toujours *.*, qui signifie *tous les fichiers*.

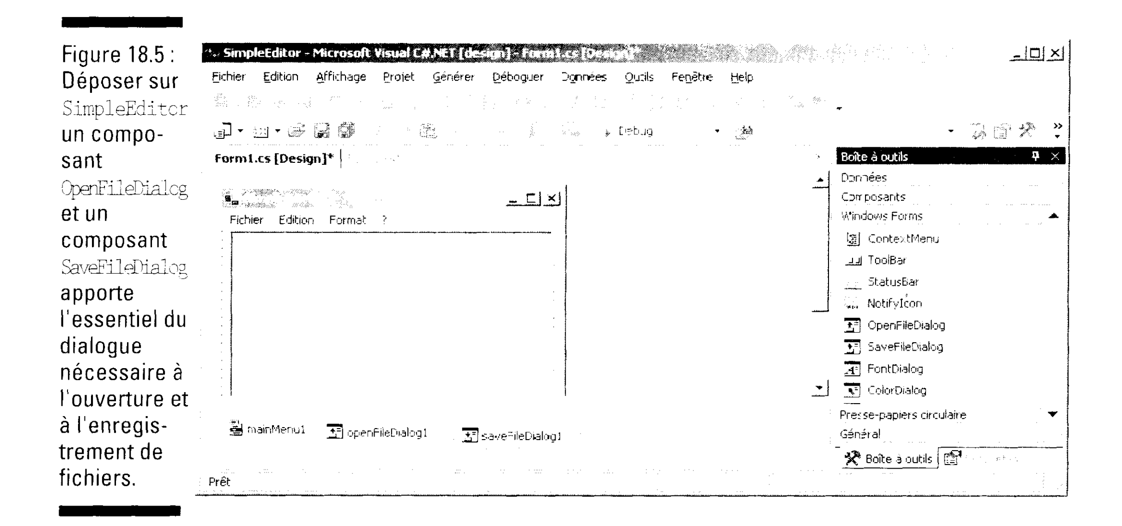
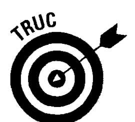


Figure 18.5 : Déposer sur SimpleEditor un composant OpenFileDialog et un composant SaveFileDialog apporte l'essentiel du dialogue nécessaire à l'ouverture et à l'enregistrement de fichiers.

Les filtres du composant `OpenFileDialog` sont stockés dans sa propriété `Filter`. La syntaxe utilisée dans ce champ est un peu déconcertante : un filtre est défini par un couple formé de son nom et de l'extension de fichier correspondante, séparés par un caractère |, deux filtres consécutifs étant également séparés par un caractère |.

3. **Assignez à la propriété `Filter` la chaîne "Fichiers RTF|*.rtf|Tous les fichiers|*.*".**

   Cette opération indique à `OpenFileDialog` de ne rechercher initialement que les fichiers RTF, mais laisse à l'utilisateur la possibilité de rechercher tous les fichiers.



"Tous les fichiers|*.*" doit toujours être la dernière entrée de la liste des filtres. Pour l'utilisateur, c'est la sélection de la dernière chance.

4. **Faite de même pour la propriété `Filter` du composant `SaveFileDialog`.**

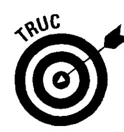
TranslationLibrary respectivement à ces deux ensembles de classes évite le problème : FileIO.Convert ne peut pas être confondu avec TranslationLibrary.Convert.

Déclarer un espace de nom

On déclare un espace de nom en utilisant le mot-clé namespace, suivi par un nom et un bloc d'accolades ouvrante et fermante. Les classes spécifiées dans ce bloc font partie de l'espace de nom.

```
namespace MyStuff
{
  class MyClass {}
  class UrClass {}
}
```

Dans cet exemple, MyClass et UrClass font partie de l'espace de nom MyStuff.



L'Assistant Application de Visual Studio place chaque classe qu'il crée dans un espace de nom portant le même nom que le répertoire qu'il crée. Examinez tous les programmes de ce livre : ils ont tous étés créés à l'aide de l'Assistant Application. Par exemple, le programme AlignOutput a été créé dans le dossier AlignOutput. Le nom du fichier source est Class1.cs, qui correspond au nom de la classe par défaut. Le nom de l'espace de nom dans lequel se trouve Class1.cs est le même que celui du dossier : AlignOutput.



Si vous ne spécifiez pas une désignation d'espace de nom, C# place votre classe dans l'espace de nom global. C'est l'espace de nom de base pour tous les autres espaces de nom.

Accéder à des modules du même espace de nom

Le nom de l'espace de nom d'une classe est une partie du nom de la classe étendue. Voyez l'exemple suivant :

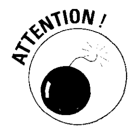
```
namespace MathRoutines
{
  class Sort
  {
    public void SomeFunction(){}
```

```
  // ignore toute erreur de conversion qui pourrait se produire
  try
  {

    // s'il y a quelque chose ici ...
    //if (sText.Length > 0)
    // ... le convertit en entier
    int nFontSize = Int32.Parse(sText);

    // si la valeur est dans l'étendue valide ...
    if (nFontSize >= trackBar1.Minimum && nFontSize <= trackBar1.Maximum)
    {
      // ... met à jour la trackbar et ...
      trackBar1.Value = nFontSize;

      // ... ajuste la police (SetFont() lit sa taille de
      // police directement sur la TrackBar
      SetFont();
    }
  }
  catch {};
}
```



Cette méthode est invoquée pour chaque caractère envoyé dans cette zone de texte, et non pour le dernier uniquement. Par exemple, la saisie de **24** génère deux appels : un pour le 2, et un pour la valeur 24. Dans cette application, c'est sans conséquence, mais sachez-le.

`FontSizeEntered()` commence par lire le contenu de `TextBox`, puis elle entre dans un bloc `try`. La fonction `Int32.Parse()` convertit le contenu de `TextBox` en une valeur `int`. Cette fonction de conversion envoie une exception si la chaîne trouvée ici ne peut pas être convertie en un entier valide. L'instruction `catch` universelle placée en bas de cette fonction bloque les problèmes, mais ignore l'exception et ne modifie pas la taille de la police. De même, si la chaîne entrée par l'utilisateur peut être convertie en un nombre entier mais que celui-ci se trouve en dehors de l'étendue autorisée (8 à 24 points), elle est ignorée.

Si la taille saisie est autorisée, `FontSizeEntered()` met à jour `TrackBar` en assignant cette nouvelle valeur à sa propriété `Value`. Par exemple, supposez que l'index de `TrackBar` soit situé à la valeur 12. Dès que l'utilisateur entre la chaîne 22, l'index passe directement à la position 22, presque à l'extrémité droite de la barre.

La fonction `Font()` met à jour la police elle-même.

ne peut pas être modifié par deux programmeurs en même temps. Chacun d'eux a besoin de son propre fichier source. Enfin, la compilation d'un module de grande taille peut prendre beaucoup de temps (on peut toujours aller prendre un café, mais il arrive un moment où votre patron devient soupçonneux). Recompiler un tel module parce qu'une seule ligne d'une seule classe a été modifiée devient intolérable.

Pour toutes ces raisons, un bon programmeur C# divise son programme en plusieurs fichiers source .CS, qui sont compilés et générés ensemble afin de former un seul exécutable.

Imaginez un système de réservation de billets d'avion : il y a l'interface avec les agents de réservation que les clients appellent au téléphone, une autre interface pour la personne qui est au comptoir d'enregistrement, la partie Internet, sans parler de la partie qui vérifie l'occupation des sièges dans l'avion, plus la partie qui calcule le prix (y compris les taxes), et ainsi de suite. Un programme comme celui-ci devient énorme bien avant d'être terminé.

Rassembler toutes ces classes dans un même fichier source `Class1.cs` est remarquablement déraisonnable, pour les raisons suivantes :

- Un fichier source ne peut être modifié que par une seule personne à la fois. Vous pouvez avoir vingt à trente programmeurs travaillant en même temps sur un grand projet. Un seul fichier pour vingt-quatre programmeurs impliquerait que chacun d'eux ne pourrait travailler qu'une heure par jour, à supposer qu'ils se relaient vingt-quatre heures sur vingt-quatre. Si vous divisiez le programme en vingt-quatre fichiers, il serait possible, bien que difficile, que tous les programmeurs travaillent en même temps. Mais si vous divisez le programme de telle manière que chaque classe a son propre fichier, l'orchestration du travail de ces vingt-quatre programmeurs devient beaucoup plus facile.

- Un fichier source unique peut devenir extrêmement difficile à comprendre. Il est beaucoup plus aisé de saisir le contenu d'un module comme `ResAgentInterface.cs`, `GateAgentInterface.cs`, `ResAgent.cs`, `GateAgent.cs`, `Fare.cs` ou `Aircraft.cs`.

- La régénération complète d'un grand programme comme un système de réservation de billets d'avion peut prendre beaucoup de temps. Vous n'aurez certainement pas envie de régénérer toutes les instructions qui composent le système simplement parce qu'un programmeur a modifié une seule ligne. Avec un programme divisé en plusieurs fichiers, Visual Studio peut régénérer uniquement le fichier modifié, et rassembler ensuite tous les fichiers objet.

1. **Dans le Concepteur de formulaires, sélectionnez la** `TrackBar`**.**

2. **Dans la fenêtre Propriétés, sélectionnez l'événement** `Scroll`**. Comme nom de fonction, entrez** FontSizeControl.

3. **Passez dans l'affichage du code source, et implémentez les nouvelles fonctions comme suit :**

```
// invoquée quand l'utilisateur déplace l'index de la TrackBar
private void FontSizeControl(object sender, System.EventArgs e)
{
  // lit la nouvelle taille de police directement sur la TrackBar
  fontSize = trackBar1.Value;

  // la convertit en chaîne et la copie dans
  // la TextBox pour accorder les deux
  textBox1.Text = String.Format("{0}", fontSize);

  // ajuste maintenant la police
  SetFont();
}
```

   `FontSizeControl()` est invoquée chaque fois que l'utilisateur déplace l'index dans la barre. Cette fonction lit la nouvelle valeur (un nombre entier compris entre 8 et 24, inclusivement) dans l'objet `TrackBar`. `FontSizeControl()` utilise la méthode `String.Format()` pour convertir ce nombre en une chaîne de texte qui est alors copiée dans la `TextBox` Taille de police. Cela fait, `FontSizeControl()` invoque `SetFont()` pour modifier la police utilisée dans la fenêtre d'édition.



   Je donne la description de `SetFont()` dans la section "Changer de police et de taille", plus haut dans ce chapitre.

4. **Générez le programme et exécutez-le.**

5. **Entrez du texte dans la fenêtre et sélectionnez-le avec la souris.**

6. **Faites glisser vers la droite et vers la gauche l'index de la taille de police.**

La Figure 18.4 montre le résultat. Il est vrai qu'une simple image n'est pas très parlante, mais la valeur qui apparaît dans la zone de texte Taille de police est instantanément mise à jour selon la position de l'index dans la barre, pendant que le texte sélectionné est agrandi ou réduit en conséquence.

Le programme `CustomException` donne la sortie suivante :

```
Erreur fatale inconnue :
Le message est <Impossible d'inverser 0>, l'objet est (Value = 0)
CustomException.MathClass
Exception envoyée parDouble Inverse()
Appuyez sur Entrée pour terminer...
```

Jetons un coup d'œil à cette sortie : le message `Erreur fatale inconnue :` vient de `Main()`. La chaîne `Le message est <Impossible d'inverser 0>, l'objet est <~~>` vient de `CustomException`. Le message `Value = 0` vient de l'objet `MathClass` lui-même. La dernière ligne, `Exception envoyée parDouble Inverse`, vient de `CustomException`.

ToString(), la carte de visite de la classe

Toutes les classes héritent d'une classe de base commune, judicieusement nommé `Object`. C'est au Chapitre 17 que j'explore cette propriété qui unifie les classes, mais il est utile de mentionner ici que `Object` contient une méthode, `ToString()`, qui convertit en `string` le contenu de l'objet. L'idée est que chaque classe doit redéfinir la méthode `ToString()` par une méthode lui permettant de s'afficher elle-même d'une façon pertinente. Dans le chapitre précédent, j'ai utilisé la méthode `GetString()` parce que je ne voulais pas y aborder les questions d'héritage, mais le principe est le même. Par exemple, une méthode `Student.ToString()` pourrait afficher le nom et le numéro d'identification de l'étudiant.

La plupart des fonctions, même les fonctions intégrées de la bibliothèque C#, utilisent la méthode `ToString()` pour afficher des objets. Ainsi, le remplacement de `ToString()` a pour effet secondaire très utile que l'objet sera affiché dans son propre format, quelle que soit la fonction qui se charge de l'affichage.

Comme dirait Bill Gates, "C'est cool."

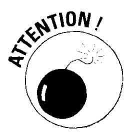
4. **Implémentez les nouvelles fonctions de la façon suivante :**

```
private void FormatBold(object sender, System.EventArgs e)
{
  isBolded = !isBolded;
  menuItem10.Checked = isBolded;
  SetFont();
}

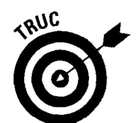
private void FormatItalics(object sender, System.EventArgs e)
{
  isItalics = !isItalics;
  menuItem11.Checked = isItalics;
  SetFont();
}
```

Chacune de ces fonctions inverse l'état de l'indicateur correspondant, et invoque `SetFont()` pour modifier la police en conséquence.

`FormatBold()` et `FormatItalics()` assignent `true` ou `false` à `menuItem10.Checked` pour placer ou non une coche devant l'option de menu correspondante afin de montrer si elle est activée (`menuItem10` correspond à l'option de menu Format/Gras, et `menuItem11` à l'option Format/Italique).



Les noms de vos options de menu pourront être différents si vous ne les avez pas mis dans le même ordre que moi. Pour savoir quel est le nom d'un composant particulier, sélectionnez celui-ci dans le Concepteur de formulaires. Le nom et le type du composant apparaissent en haut de la fenêtre Propriétés.



Rien ne vous oblige à vous en tenir aux noms assignés par le Concepteur de formulaires. Vous pouvez changer à votre guise la propriété `Name` lorsque vous créez l'objet. Vous pouvez ainsi choisir des noms plus parlants et plus faciles à retenir.

5. **Par acquit de conscience, ajoutez dans le constructeur un appel à `SetFont()` pour définir correctement la police initiale :**

```
public Form1()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();
```

```
    //           le message standard Exception.ToString()
    override public string ToString()
    {
      string s = Message + "\n";
      s += base.ToString();
      return s;
    }
    // Inverse - retourne 1/x
    public double Inverse()
    {
      if (nValueOfObject == 0)
      {
        throw new CustomException("Impossible d'inverser 0", this);
      }
      return 1.0 / (double)nValueOfObject;
    }
}
  public class Class1
  {
    public static void Main(string[] args)
    {
      try
      {
        // prend l'inverse de 0
        MathClass mathObject = new MathClass("Valeur", 0);
        Console.WriteLine("L'inverse de d.Value est{0}",
                          mathObject.Inverse());
      }
      catch(Exception e)
      {
        Console.WriteLine("\nErreur fatale inconnue :\n{0}",
                          e.ToString());
      }

      // attend confirmation de l'utilisateur
      Console.WriteLine("Appuyez sur Entrée pour terminer...");
      Console.Read();
    }
  }
}
```

Permettez-moi de faire une remarque : cette classe `CustomException` n'est pas si remarquable que cela. Elle stocke un message et un objet, tout comme `MyException`. Toutefois, au lieu de fournir de nouvelles méthodes pour accéder à ces données, elle remplace la propriété `Message` existante qui retourne le message d'erreur contenu dans l'exception, et la méthode `ToString()` qui retourne le message plus l'indication de pile.

SimpleEditor. Le format RTF conserve les informations sur la mise en forme : la première ligne était en style Titre 1, la deuxième en style Normal, et la troisième en style Code.

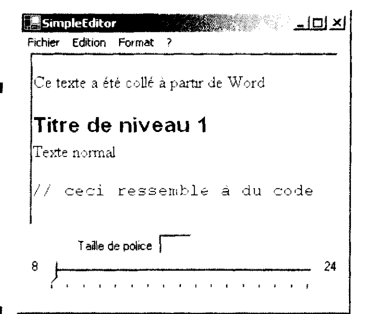


Figure 18.2 : Le texte venant de Word a conservé sa mise en forme.



C'est le composant RichTextBox qui fait tout le travail de mise en forme qu'il fallait autrefois faire soi-même au prix de mille difficultés.

Mettre hardiment en gras et en italique

SimpleEditor peut maintenant couper et coller du texte mis en forme, mais il ne sait pas encore modifier la police. Pour cela, il nous faut introduire les contrôles du menu Format et du champ Taille de police.

Changer de police et de taille

Changer de police, mettre le texte en gras ou en italique est en fait presque la même fonction. Vous pouvez donc vous épargner quelques efforts en créant une seule fonction capable de réaliser ces trois opérations, comme ceci :

```
//---------------Mise en forme et taille de la police-------------
bool isBolded = false;
bool isItalics = false;
float fontSize = 12.0F;
private void SetFont()
{
  FontStyle fs = FontStyle.Regular;
  if (isBolded)
  {
```

Renvoyer le même objet exception présente un avantage et un inconvénient. Cela permet aux fonctions intermédiaires d'attraper des exceptions pour libérer ou fermer des éléments alloués par elles, tout en permettant à l'utilisateur final de l'objet exception de suivre l'indication de pile jusqu'à la source de l'exception. Toutefois, une fonction intermédiaire ne peut pas (ou ne doit pas) ajouter des informations supplémentaires à l'exception en la modifiant avant de la renvoyer.

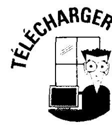
Redéfinir une classe d'exceptions

La classe d'exceptions suivante définie par l'utilisateur peut stocker des informations supplémentaires qui ne pourraient pas l'être dans un objet Exception conventionnel :

```
// MyException - ajoute à la classe standard Exception
//              une référence à MyClass
public class MyException : Exception
{
  private MyClasss myobject;
  MyException(string sMsg, MyClass mo) : base(sMsg)
  {
    myobject = mo;
  }
  // permet aux classes extérieures d'accéder à une classe d'information
  public MyClass MyObject{ get {return myobject;}}
}
```

Voyez à nouveau ma bibliothèque de fonctions BrilliantLibrary. Ces fonctions savent comment remplir ces nouveaux membres de la classe MyException et aller les chercher, fournissant ainsi uniquement les informations nécessaires pour remonter à la source de toute erreur connue et de quelques autres restant à découvrir. L'inconvénient de cette approche est que seules les fonctions de la bibliothèque BrilliantLibrary peuvent recevoir un bénéfice quelconque des nouveaux membres de MyException.

Le remplacement des méthodes déjà présentes dans la classe Exception peut donner des fonctions existantes autres que l'accès BrilliantLibrary aux nouvelles données. Considérez la classe d'exceptions définie dans le programme CustomException suivant :




```
// CustomException - crée une exception personnalisée qui
//                   affiche les informations que nous voulons, mais
//                   dans un format plus agréable
```

```
  // et voilà, nous avons ce qu'il nous faut
  return (string)o;
}
```

`ReadClipboard()` commence par essayer de récupérer un objet dans le Presse-papiers. Si elle n'y trouve rien, elle ne retourne rien. Elle essaie alors de lire dans l'objet une chaîne RTF. Encore une fois, si l'objet `clipboard` n'est pas une chaîne RTF, elle retourne `null`. Enfin, s'il y a quelque chose dans le Presse-papiers, et si c'est une chaîne RTF, `ReadClipboard()` retourne la chaîne à la fonction appelante.

1. **Dans l'Explorateur de solutions, double-cliquez sur `Form1.cs` pour afficher le code source C# du projet.**

2. **Dans le code source, insérez les méthodes `WriteClipboard()` et `ReadClipboard()`.**

   Utilisez pour cela Édition/Couper, Édition/Copier, et Édition/Coller.

3. **Dans le Concepteur de formulaires, sélectionnez l'option de menu Édition/Coller.**

4. **Dans la fenêtre Propriétés, cliquez sur le bouton contenant un éclair pour afficher les propriétés actives (les événements). Sélectionnez l'événement `Click`.**

5. **Dans le champ qui se trouve à droite de `Click`, entrez un nom de fonction significatif. Pour que ce soit cohérent avec l'option de menu, j'ai mis `EditPaste`.**

   Le Concepteur de formulaires crée une méthode vide, qui est liée à l'option de menu de telle sorte que c'est cette méthode qui est invoquée lorsque l'utilisateur clique sur celui-ci.

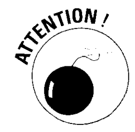
6. **Répétez les étapes 3 à 5 pour les options de menu Couper et Copier. J'ai nommé ces méthodes `EditCut` et `EditCopy`.**

7. **Vous devez ajouter manuellement le contenu de ces méthodes, comme suit :**

```
private void EditCut(object sender, System.EventArgs e)
{
  string rtfText = richTextBox1.SelectedRtf;
  WriteClipboard(rtfText);
```

types d'exception définie pour la brillante bibliothèque de classe que je viens d'écrire (c'est pour ça que je l'appelle `BrilliantLibrary`). Les fonctions qui composent `BrilliantLibrary` envoient et attrapent des exceptions `MyException`.

Toutefois, les fonctions de la bibliothèque `BrilliantLibrary` peuvent aussi appeler des fonctions de la bibliothèque générique `System`. Les premières peuvent ne pas savoir comment traiter les exceptions de la bibliothèque `System`, en particulier si elles sont causées par une entrée erronée.



Si vous ne savez pas quoi faire avec une exception, laissez-la passer pour qu'elle arrive à la fonction appelante. Mais soyez honnête avec vous-même : ne laissez pas passer une exception parce que vous n'avez simplement pas le courage d'écrire le code de traitement d'erreur correspondant.

Relancer un objet

Dans certains cas, une méthode ne peut pas traiter entièrement une erreur, mais ne veut pas laisser passer l'exception sans y mettre son grain de sel. C'est comme une fonction mathématique qui appelle `Factorial()` pour s'apercevoir qu'elle renvoie une exception. Même si la cause première du problème peut être une donnée incorrecte, la fonction mathématique est peut-être en mesure de fournir des indications supplémentaires sur ce qui s'est passé.

Un bloc `catch` peut digérer partiellement l'exception envoyée et ignorer le reste. Ce n'est pas ce qu'il y a de plus beau, mais ça existe.

L'interception d'une exception d'erreur est une chose très courante pour les méthodes qui allouent des éléments. Par exemple, imaginez une méthode `F()` qui ouvre un fichier quand elle est invoquée, et le referme quand elle se termine. Quelque part dans le cours de son exécution, `F()` invoque `G()`. Une exception envoyée de `G()` passerait directement à travers `F()` sans lui laisser la moindre chance de fermer le fichier. Celui-ci resterait donc ouvert jusqu'à ce que le programme lui-même se termine. Une solution idéale serait que `F()` contienne un bloc `catch` qui ferme les fichiers ouverts. Bien entendu, `F()` est libre de passer l'exception au niveau supérieur après en avoir fait ce qu'il fallait pour ce qui la concerne.

Il y a deux manières de renvoyer une erreur. La première consiste à envoyer une deuxième exception, avec les mêmes informations ou éventuellement des informations supplémentaires :

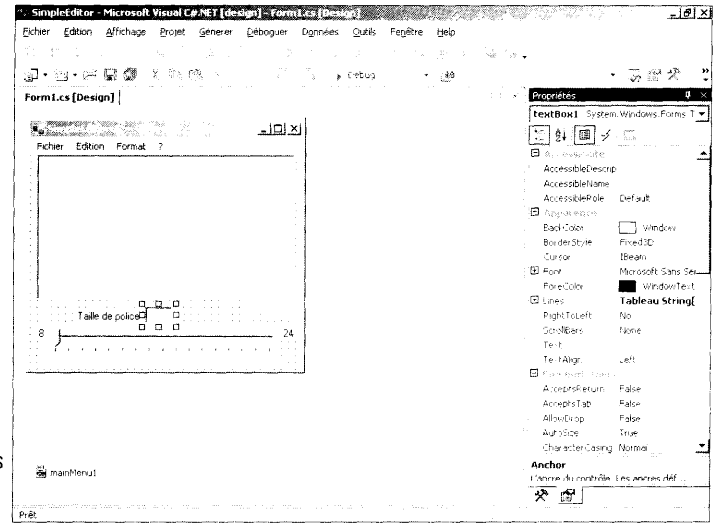


Figure 18.1 : Même le modeste composant `TextBox` a des dizaines de propriétés actives.

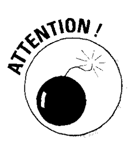
Un menu garanti pour éditer le menu Édition

Comme il n'y a aucune raison particulière de commencer par un composant plutôt que par un autre, pourquoi ne pas aller du plus facile au plus difficile ? Dans les premières versions de Windows, l'une des fonctions les plus difficiles à manier était le Presse-papiers, mais la bibliothèque de C# fait de la copie et de la récupération de données dans le Presse-papiers un jeu d'enfant.



Le Presse-papiers est une zone d'arrière-plan de Windows dans laquelle est stocké ce qui a été copié ou coupé en attendant d'être collé par la suite. C'est dans Windows que doit se trouver le Presse-papiers, parce que l'utilisateur peut y copier un objet, par exemple du texte, dans une application, et le coller dans une autre.

La méthode `SetDataObject` de la classe `Clipboard` (le *clipboard* est le Presse-papiers) écrit un objet `DataObject` dans le Presse-papiers. L'objet `DataObject` contient les données à stocker et la description de leur type.



L'identification du type de donnée est très importante. Par exemple, l'utilisateur peut essayer de couper le contenu d'une feuille de calcul et de le coller dans la fenêtre de `SimpleEditor`. S'il n'y avait pas un moyen de filtrer ce contenu, `SimpleEditor` afficherait une chaîne de n'importe quoi (au mieux).

```
      Console.WriteLine(e.Message);
    }
  }

  // f2 -  - préparez-vous à attraper une exception MyException
  public void f2(bool bExceptionType)
  {
    try
    {
      f3(bExceptionType);
    }
    catch(MyException me)
    {
      Console.WriteLine("Exception MyException attrapée dans f2()");
      Console.WriteLine(me.Message);
    }
  }

  // f3 -  - n'essayez pas d'attraper des exceptions
  public void f3(bool bExceptionType)
  {
    f4(bExceptionType);
  }

  // f4 -  - envoie des exceptions d'un type ou d'un autre
  public void f4(bool bExceptionType)
  {
    // nous travaillons avec un objet local
    MyClass mc = new MyClass();
    if(bExceptionType)
    {
      // une erreur se produit - l'objet est envoyé avec l'exception
      throw new MyException("MyException envoyée dans f4()",
                            mc);
    }
    throw new Exception("Exception générique envoyée dans f4()");
  }

  public static void Main(string[] args)
  {
    // envoie d'abord une exception générique
    Console.WriteLine("Envoie d'abord une exception générique");
    new Class1().f1(false);
    // envoie maintenant une de mes exceptions
    Console.WriteLine("\nEnvoie d'abord une exception spécifique");
    new Class1().f1(true);

    // attend confirmation de l'utilisateur
    Console.WriteLine("Hit Appuyez sur Entrée pour terminer...");
```

Chapitre 18

Achever votre application Windows

Dans ce chapitre :

Implémenter les options des menus.

Copier des données dans le Presse-papiers et les récupérer.

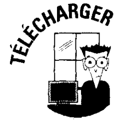
Faire des manipulations simples sur les polices.

Lier ensemble deux contrôles pour que les modifications effectuées dans l'un soient reflétées dans l'autre.

Lire et enregistrer le contenu de l'éditeur.

Utiliser des boîtes de dialogue.

Au Chapitre 17, nous avons créé une jolie petite application `SimpleEditor`. Malheureusement, au point où nous en sommes, elle ne fait encore rien. Dans ce chapitre, nous allons ajouter les couches nécessaires pour la faire fonctionner. Vous allez transformer `SimpleEditor` en quelque chose d'utile : un éditeur capable de lire et d'écrire des fichiers texte, de mettre du texte en gras et en italique, de changer la taille de la police, d'écrire dans le Presse-papiers et d'en lire le contenu, et que l'on peut redimensionner à volonté.



Le code complet de `SimpleEditor` est sur le site Web.

Ajouter des actions

Le Concepteur de formulaires simplifie le travail de création d'une application Windows. Il permet d'ouvrir une Boîte à outils regorgeant d'accessoires comme des boutons, des zones de texte ou des étiquettes (de façon plus

```
    }
    catch(Exception e)
    {
      // les autres exceptions non encore attrapées sont attrapées ici
    }
  }
```

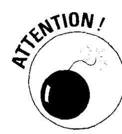
Si SomeOtherFunction() envoyait un objet Exception, celui-ci ne serait pas attrapé par l'instruction catch(MyException) car une Exception n'est pas de type MyException. Il serait attrapé par l'instruction catch suivante : catch(Exception).



Toute classe qui hérite de MyException EST_UNE MyException :

```
class MySpecialException : MyException
{
  // . . . instructions quelconques ...
}
```

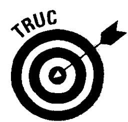
Si elle en a la possibilité, l'instruction catch MyException attrapera tout objet MySpecialException envoyé.



Faites toujours se succéder les instructions catch de la plus spécifique à la plus générale. Ne placez jamais en premier l'instruction catch la plus générale :

```
public void SomeFunction()
{
  try
  {
    SomeOtherFunction();
  }
  catch(Exception me)
  {
    // tous les objets MyException sont attrapés ici
  }
  catch(MyException e)
  {
    // aucune exception ne parvient jamais jusqu'ici parce qu'elle
    // est attrapée par une instruction catch plus générale
  }
}
```

Dans cet exemple, l'instruction catch la plus générale coupe l'herbe sous le pied de la suivante en interceptant tous les envois.



Les propriétés que vous n'avez pas modifiées sont les propriétés par défaut de l'objet, qui n'apparaissent donc pas dans le code. Par exemple, vous pouvez voir que `trackBar1` est ancrée sur `AnchorStyles.Bottom`, `AnchorStyles.Left` et `AnchorStyles.Right`. D'autre part, les propriétés `Maximum` et `Minimum` reçoivent les valeurs 24 et 8, et la propriété `Value`, la taille de police initiale, reçoit la valeur 12.

Le reste de la méthode `InitializeComponents()` est très long, mais suit la même logique qui consiste à assigner une valeur à chaque propriété modifiée.

Comment apprendre à connaître les composants ?

L'une des questions souvent posées par les nouveaux programmeurs C# pour Windows est : "Comment sait-on quels composants sont disponibles et ce que fait chacun d'eux ?" Un bon moyen de faire connaissance consiste à jouer avec eux : choisissez un composant, faites-le glisser sur le formulaire, sélectionnez-le, et commencez à passer en revue ses propriétés.

Un autre moyen consiste à rechercher les composants dans l'aide de Visual Studio. Le nom de la classe est le même que celui qui apparaît dans la Boîte à outils. Ainsi, si vous voulez savoir comment utiliser un `RadioButton`, vous pouvez commencer par entrer **RadioButton** dans l'index de l'aide. L'aide de Visual Studio fait apparaître une fenêtre contenant une description du composant, et parfois un exemple de code.

Enfin, dans tous les exemples que vous pourrez trouver, explorez la méthode `InitializeComponents()` jusqu'à ce que vous arriviez à comprendre ce qu'a fait le Concepteur de formulaires. Ce procédé vous permettra de découvrir de nouveaux composants et leurs propriétés, et vous donnera une idée de ce à quoi ils servent.

Avec l'expérience, il devient de plus en plus facile de trouver le bon composant.

Et maintenant ?

N'oubliez pas que le programme `SimpleEditor` que nous avons créé dans ce chapitre est très simple. Il est très joli, mais il est si simple qu'en réalité qu'il ne fait rien. Au Chapitre 18, nous allons ajouter le code nécessaire pour faire de `SimpleEditor` un véritable éditeur.

Cette classe CustomException est faite sur mesure pour signaler une erreur au logiciel qui traite avec la tristement célèbre MyClass. Cette sous-classe d'Exception met de côté la même chaîne que l'original, mais dispose en plus de la possibilité de stocker dans l'exception la référence au fautif.

L'exemple suivant attrape la classe CustomException et met en utilisation ses informations sur MyClass :

```
public class Class1
{
  public void SomeFunction()
  {
    try
    {
      // . . . opérations préalables à la fonction exemple
      SomeOtherFunction();
      // . . . autres opérations. . .
    }
    catch(MyException me)
    {
      // vous avez toujours accès aux méthodes d'Exception
      string s = me.ToString();
      // mais vous avez aussi accès à toutes les propriétés et méthodes
      // de votre propre classe d'exceptions
      MyClass mo = me.MyCustomObject;
      // par exemple, demandez à l'objet MyClass de s'afficher lui-même
      string s = mo.GetDescription();
    }
  }
  public void SomeOtherFunction()
  {
    // création de myobject
    MyClass myobject = new MyClass();
    // . . . signale une erreur concernant myobject . . .
    throw new MyException("Erreur dans l'objet de MyClass", myobject);
    // . . . reste de la fonction . . .
  }
}
```

Dans ce fragment de code, SomeFunction() invoque SomeOtherFunction() de l'intérieur d'un bloc try. SomeOtherFunction() crée et utilise un objet myobject. Quelque part dans SomeOtherFunction(), une fonction de vérification d'erreur se prépare à envoyer une exception pour signaler qu'une condition d'erreur vient de se produire. Plutôt que de créer une simple Exception, SomeFunction() se sert de la toute nouvelle classe MyException, pour envoyer non seulement un message d'erreur, mais aussi l'objet myobject fautif.

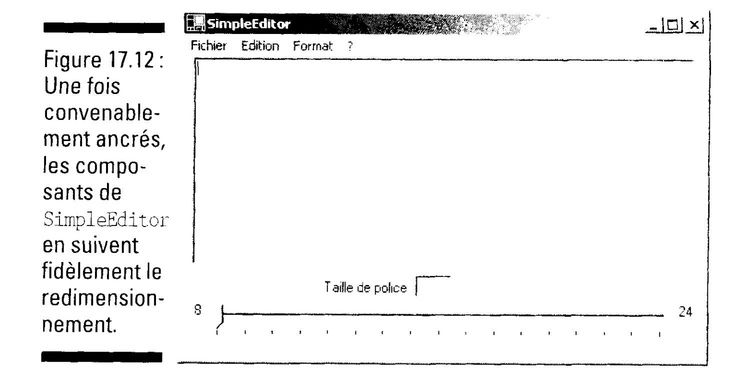
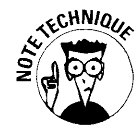


Figure 17.12 : Une fois convenablement ancrés, les composants de `SimpleEditor` en suivent fidèlement le redimensionnement.

Qu'avons-nous fabriqué ?

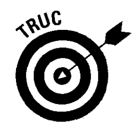
Le listing ci-dessous montre un sous-ensemble de la méthode `InitializeComponent()` créée par le Concepteur de formulaires. Puisque cette méthode est très volumineuse, je n'en donne ici qu'un petit extrait :

```
namespace SimpleEditor
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
  {
    private System.Windows.Forms.RichTextBox richTextBox1;
    private System.Windows.Forms.MainMenu mainMenu1;
    private System.Windows.Forms.MenuItem menuItem1;
    //...
    #region Windows Form Designer generated code
    private void InitializeComponent()
    {
      this.mainMenu1 = new System.Windows.Forms.MainMenu();
      this.menuItem1 = new System.Windows.Forms.MenuItem();
      this.trackBar1 = new System.Windows.Forms.TrackBar();
      //....
      //
      // mainMenu1
      //
      this.mainMenu1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {
                                                            this.menuItem1,
```



Comme Main() est le point de départ du programme, il est bon de toujours en placer le contenu dans un bloc try. Toute exception qui ne sera pas "attrapée" ailleurs remontera finalement jusqu'à Main(). C'est donc votre dernière opportunité de récupérer une erreur avant qu'elle aboutisse à Windows, dont le message d'erreur sera beaucoup plus difficile à interpréter.

Le bloc catch situé à la fin de Main() attrape l'objet Exception et utilise sa méthode ToString() pour afficher sous forme d'une simple chaîne la majeure partie des informations sur l'erreur contenues dans l'objet exception.



La propriété Exception.Message retourne un sous-ensemble plus lisible, mais moins descriptif des informations sur l'erreur.

Cette version de la fonction Factorial() contient la même vérification pour un argument négatif que la précédente. Si l'argument est négatif, Factorial() met en forme un message d'erreur qui décrit le problème, incluant la valeur incriminée. Factorial() regroupe ensuite ces informations dans un objet Exception nouvellement créé, qu'elle envoie à la fonction appelante.

La sortie de ce programme apparaît comme suit (j'ai un peu arrangé les messages d'erreur pour les rendre plus lisibles) :

```
i = 6, factorielle = 720
i = 5, factorielle = 120
i = 4, factorielle = 24
i = 3, factorielle = 6
i = 2, factorielle = 2
i = 1, factorielle = 1
i = 0, factorielle = 0
Erreur fatale :
System.Exception: Argument négatif illicite passé à Factorial -1
   at Factorial(Int32 nValue) in c:\c#program\Factorial\class1.cs:line 23
   at FactorialException.Class1.Main(String[] args) in c:\c#program\Factorial\
   class1.cs:line 52
Appuyez sur Entrée pour terminer...
```

Les premières lignes affichent les véritables factorielles des nombres 6 à 0. La factorielle de -1 génère un message commençant par Erreur fatale, ce qui est susceptible d'attirer l'attention de l'utilisateur.

La première ligne du message d'erreur a été mise en forme dans la fonction Factorial() elle-même. Cette ligne décrit la nature du problème, en indiquant la valeur incriminée -1.

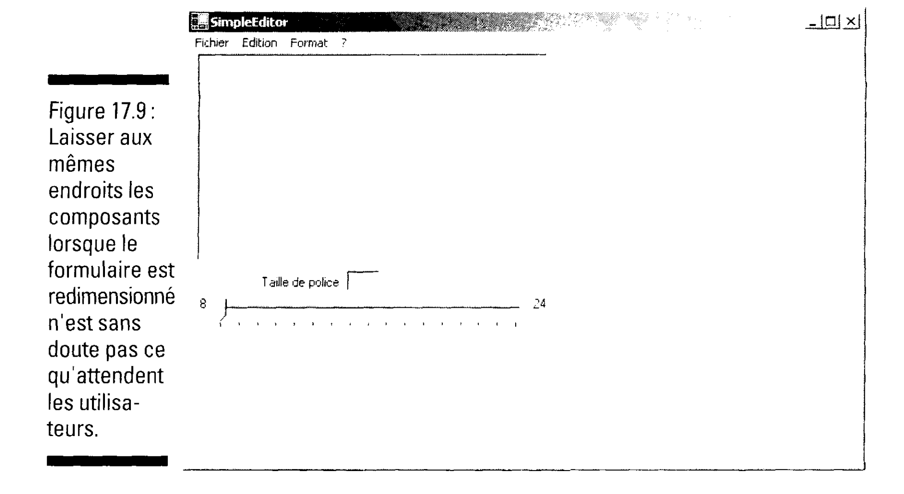


Figure 17.9 : Laisser aux mêmes endroits les composants lorsque le formulaire est redimensionné n'est sans doute pas ce qu'attendent les utilisateurs.

En somme, la `TrackBar` doit être ancrée aux bords inférieur, droit et gauche du formulaire.

2. **Pour définir l'ancrage, sélectionnez la `TrackBar`, et cliquez sur `Anchor` dans la fenêtre Propriétés.**

   La propriété d'ancrage apparaît sur la droite, avec une petite flèche pointant vers le bas.

3. **Cliquez sur la flèche.**

   Une petite fenêtre apparaît, contenant quatre bras formant une croix, chacun d'eux représentant un ancrage. Vous pouvez voir que l'ancrage par défaut est le coin supérieur gauche du formulaire (ce qui explique pourquoi la `TrackBar` ne bougeait pas quand on redimensionnait le formulaire).

4. **Sélectionnez les bras du bas, de droite et de gauche, et désélectionnez celui du haut.**

   La Figure 17.10 montre le résultat.

   TRUC

   Si vous préférez, vous pouvez aussi taper manuellement **Bottom**, **Top**, **Left**, ou **Right** dans le champ `Anchor` sans utiliser la fenêtre d'ancrage.

5. **Définissez l'ancrage séparément pour chaque composant.**

   Le Tableau 17.2 indique l'ancrage qui convient pour chaque composant.

```
        throw new Exception("Description de l'erreur");
        // . . . suite de la fonction . . .
    }
  }
```

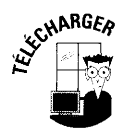
La fonction `SomeFunction()` contient un bloc de code identifié par le mot-clé `try`. Toute fonction appelée dans ce bloc, ou toute fonction qui l'appelle, est considérée comme faisant partie du bloc `try`.

Un bloc `try` est immédiatement suivi par le mot-clé `catch`, lequel est suivi par un bloc auquel le contrôle est passé si une erreur se produit en un endroit quelconque dans le bloc `try`. L'argument passé au bloc `catch` est un objet de la classe `Exception` ou d'une sous-classe de celle-ci.

À un endroit quelconque dans les profondeurs de `SomeOtherFunction()`, une erreur se produit. Toujours prête, la fonction signale une erreur à l'exécution en envoyant (`throw`) un objet `Exception` au premier bloc pour que celui-ci l'attrape (`catch`).

Puis-je avoir un exemple ?

Le programme `FactorialException` suivant met en évidence les éléments clés du mécanisme des exceptions :

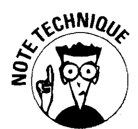



```
// FactorialException - crée une fonction factorielle qui
//                      indique à Factorial() les arguments illicites
//                      en utilisant un objet Exception
using System;
namespace FactorialException
{
  // MyMathFunctions - collection de fonctions mathématiques
  //                   de ma création (pas encore grand-chose à montrer)
  public class MyMathFunctions
  {
    // Factorial - retourne la factorielle d'une valeur
    //             fournie
    public static double Factorial(int nValue)
    {
      // interdit les nombres négatifs
      if (nValue < 0)
      {
        // signale un argument négatif
        string s = String.Format(
             "Argument négatif illicite passé à Factorial {0}",
```

1. **Pour résoudre ce problème, placez un composant `Label` à gauche de la `TextBox`, et entrez "Taille de police" dans sa propriété `Text`. Une police Arial en 10 points gras conviendra bien à l'étiquette de la `TextBox`.**

2. **Ajoutez maintenant une étiquette à l'extrémité gauche de la `TrackBar`. Donnez-lui également une police Arial en 10 points gras. Entrez 8 dans la propriété `Text`, puisque c'est la valeur que nous avons donnée à la propriété `Minimum` de la `TrackBar`.**

3. **Faites de même pour l'étiquette de l'extrémité droite de la `TrackBar`. Entrez 24 dans sa propriété Texte, qui est la valeur de la propriété `Maximum` de la `TrackBar`.**



Au Chapitre 18, je vous montrerai comment les valeurs de ces étiquettes peuvent être définies automatiquement en fonction des propriétés de la `TrackBar`, mais pour le moment nous allons nous contenter de le faire manuellement.

4. **Encore une fois, générez à nouveau `SimpleEditor` pour être sûr que tout va bien.**

Personne ne pourrait être plus heureux avec le résultat montré par la Figure 17.8.

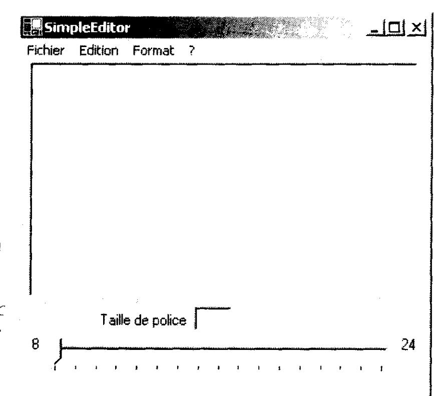


Figure 17.8 : `SimpleEditor` est prêt pour aller danser.

d'erreur que la fonction appelante ne teste pas. Bien sûr, en tant que programmeur en chef, je peux me laisser aller à proférer des menaces. Je me souviens d'avoir lu toutes sortes de livres de programmation regorgeant de menaces de bannissement du syndicat des programmeurs pour ceux qui ne s'occupent pas des codes d'erreur, mais tout bon programmeur FORTRAN sait bien qu'un langage ne peut obliger personne à vérifier quoi que ce soit, et que, très souvent, ces vérifications ne sont pas faites.

Souvent, même si je vérifie l'indication d'erreur retournée par `Factorial()` ou par toute autre fonction, la fonction appelante ne peut rien faire d'autre que de signaler l'erreur. Le problème est que la fonction appelante est obligée de tester toutes les erreurs possibles retournées par toutes les fonctions qu'elle appelle. Bien vite, le code commence à avoir cette allure là :

```
// appelle SomeFunction, lit l'erreur retournée, la traite
// et retourne
errRtn = someFunc();
if (errRtn == SF_ERROR1)
{
  Console.WriteLine("Erreur de type 1 sur appel à someFunc()");
  return MY_ERROR_1;
}
if (errRtn == SF_ERROR2)
{
  Console.WriteLine("Erreur de type 2 sur appel à someFunc()");
  return My_ERROR_2;
}
// appelle SomeOtherFunctions, lit l'erreur, retourne, et ainsi de suite
errRtn = someOtherFunc();
if (errRtn == SOF_ERROR1)
{
  Console.WriteLine("Erreur de type 1 sur appel à someFunc()");
  return MY_ERROR_3;
}
if (errRtn == SOF_ERROR2)
{
  Console.WriteLine("Erreur de type 1 sur appel à someFunc()");
  return MY_ERROR_4;
}
```

Ce mécanisme présente plusieurs inconvénients :

- Il est très répétitif.
- Il oblige le programmeur à inventer de nombreuses indications d'erreur et à en maîtriser l'emploi.

Au lieu de faire apparaître les sous-propriétés de `Font` dans la fenêtre Propriétés, vous pouvez ouvrir la boîte de dialogue Police : dans la boîte de dialogue Propriétés, cliquez sur `Font`, puis cliquez sur le petit carré gris contenant des points de suspension qui apparaît dans ce champ.

2. **Dans la fenêtre `Police`, sélectionnez la police Arial, une taille de 12 points, et le style Gras.**

3. **Faites glisser le coin inférieur gauche de la zone de texte afin de la redimensionner pour deux chiffres dans la taille et la police que vous venez de sélectionner.**

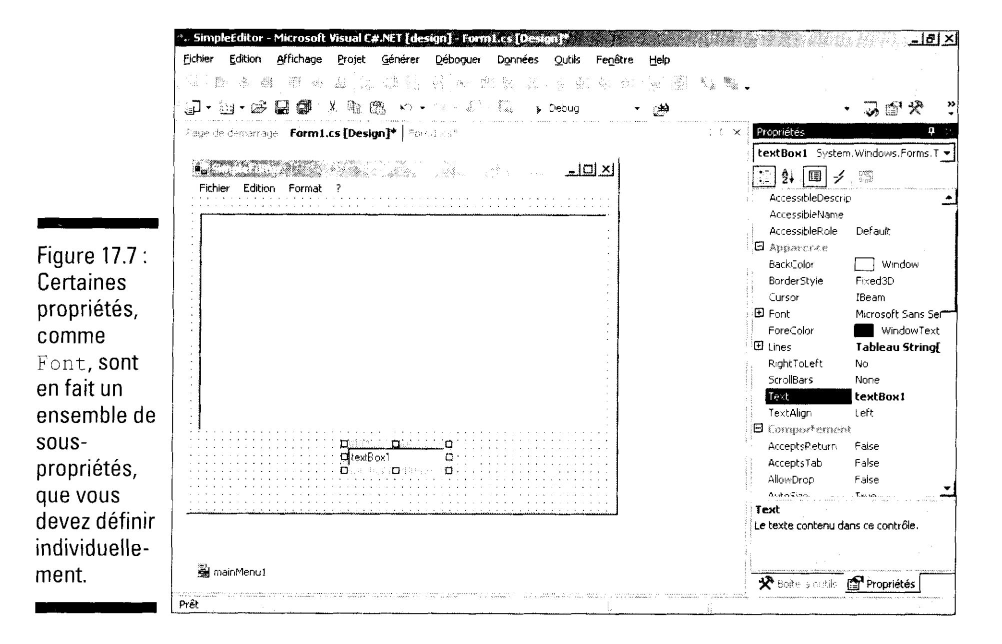
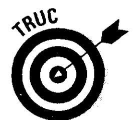


Figure 17.7 : Certaines propriétés, comme `Font`, sont en fait un ensemble de sous-propriétés, que vous devez définir individuellement.

4. **Assignez `Center` à la propriété `TextAlign`, supprimez le contenu de `Text`, et vous y êtes.**

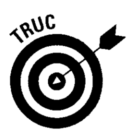


Au cas où vous ne sauriez pas très bien ce que vous permet de faire un certain composant, la liste des propriétés est là, dans la fenêtre Propriétés, pour vous en donner une idée. Sélectionnez simplement le composant, et parcourez le contenu de la fenêtre Propriétés en définissant les propriétés selon ce qui vous convient. Vous ne pouvez pas faire de dégâts : si vous n'aimez pas le résultat, vous pourrez toujours modifier la propriété. Si vous avez modifié tant de choses que vous ne vous y retrouvez plus, sélectionnez le composant, appuyez sur la touche Suppr, et il a disparu.

```
            if (dFactorial == MyMathFunctions.NON_INTEGER_VALUE)
            {
              Console.WriteLine
                        ("Factorial() a reçu un nombre non entier");
              break;
            }
            // affiche le résultat à chaque passage
            Console.WriteLine("i = {0}, factorielle = {1}",
                              i, MyMathFunctions.Factorial(i));
          }
          // attend confirmation de l'utilisateur
          Console.WriteLine("Appuyez sur Entrée pour terminer...");
          Console.Read();
        }
      }
    }
```

`Factorial()` commence maintenant par effectuer une série de tests. Le premier regarde si la valeur passée est négative (0 est accepté parce qu'il donne un résultat raisonnable). Si oui, la fonction retourne immédiatement une indication d'erreur. Si non, la valeur de l'argument est comparée à sa version entière : si elles sont égales, c'est que la partie décimale de l'argument est nulle.

`Main()` teste le résultat retourné par `Factorial()`, à la recherche de l'indication éventuelle d'une erreur. Toutefois, des valeurs comme -1 et -2 n'ont guère de signification pour un programmeur qui effectue la maintenance de son code ou qui l'utilise. Pour rendre un peu plus parlante l'erreur retournée, la classe `MyMathFunctions` définit deux constantes entières. La constante `NEGATIVE_NUMBER` reçoit la valeur -1, et `NON_INTEGER_VALUE` reçoit la valeur -2. Cela ne change rien, mais l'utilisation des constantes rend le programme beaucoup plus lisible, en particulier la fonction appelante `Main()`.



Dans la convention sur les noms Southern Naming Convention, les noms des constantes sont entièrement en majuscules, les mots étant séparés par un tiret de soulignement. Certains programmeurs, plus libéraux, refusent de faire allégeance, mais ce n'est pas la convention qui a des chances de changer.



Les constantes contenant les valeurs d'erreur sont accessibles par la classe, comme dans `MyMathClass.NEGATIVE_NUMBER`. Une variable de type `const` est automatiquement statique, ce qui en fait une propriété de classe partagée par tous les objets.

La fonction `Factorial()` signale maintenant qu'une valeur négative lui a été passée comme argument. Elle le signale à `Main()` qui se termine alors en affichant un message d'erreur beaucoup plus intelligible :

Cliquez sur l'un des éléments du menu, et examinez la liste des propriétés. On y trouve bien sûr une propriété `Text` – cette fois, c'est l'étiquette du menu. Le champ `Shortcut` est aussi une propriété intéressante, montrée par la Figure 17.6. Si vous cliquez dessus, une liste déroulante apparaît, contenant tous les raccourcis clavier que vous pouvez utiliser pour sélectionner cet élément.

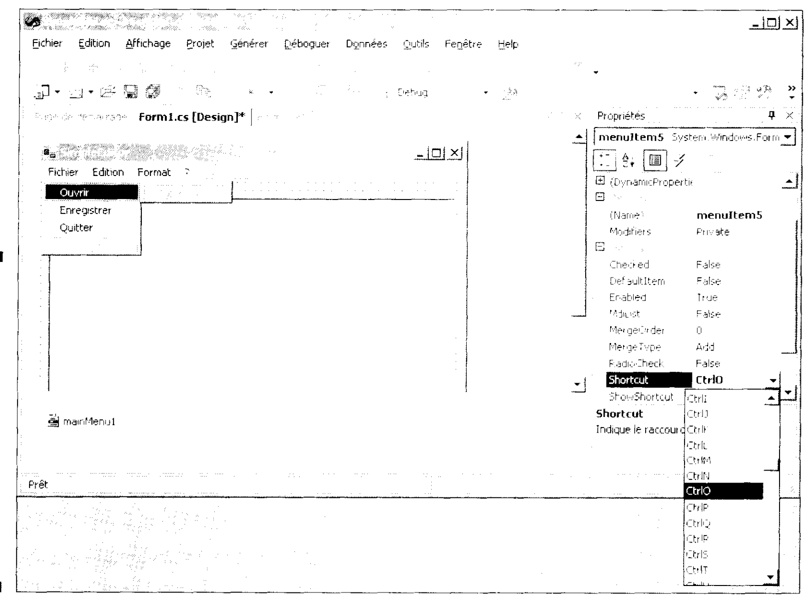


Figure 17.6 : La propriété `Shortcut` permet de spécifier le raccourci clavier que vous voulez assigner à l'élément de menu.



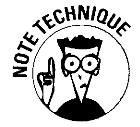
Appuyer sur un raccourci clavier permet d'exécuter la même commande qu'en sélectionnant l'option de menu associée. Par exemple, appuyer sur Ctrl+C donne le même résultat que Édition/Copier, mais c'est plus rapide.

7. **Dans le formulaire, cliquez sur l'élément de menu Fichier/Ouvrir, et assignez CtrlO à sa propriété `Shortcut`, comme le montre la Figure 17.6. En répétant ce processus, assignez un raccourci clavier à tous les autres éléments de menu, selon le Tableau 17.1.**

8. **Comme d'habitude, générez à nouveau le programme et essayez-le.**

Vous pouvez taper dans la zone de texte et ouvrir les différents menus principaux. Naturellement, au point où nous en sommes, `SimpleEditor` est encore plus que simple. C'est une coquille vide. Si vous sélectionnez Fichier/Quitter, il ne se passe rien du tout. Il vous faut ajouter du code derrière chacun de ces éléments de menu pour qu'ils fassent quelque chose. C'est le sujet du Chapitre 18.

négatif. Ensuite, remarquez que les valeurs négatives ne croissent pas de la même manière que les valeurs positives. Manifestement, il y a quelque chose qui cloche.



Les résultats incorrects retournés ici sont assez subtils par rapport à ce qui aurait pu se produire. Si la boucle de `Factorial()` avait été écrite sous la forme `do {...} while (dValue != 0)`, le programme se serait planté en passant un nombre négatif. Bien sûr, je n'aurais jamais écrit une condition comme `while(dValue != 0)`, car les erreurs dues à l'approximation auraient pu faire échouer de toute façon la comparaison avec zéro.

Retourner une indication d'erreur

Bien qu'elle soit assez simple, il manque à la fonction `Factorial()` une importante vérification d'erreur : la factorielle d'un nombre négatif n'est pas définie, pas plus que la factorielle d'un nombre non entier. La fonction `Factorial()` doit donc comporter un test pour vérifier que ces conditions sont remplies.

Mais que fera la fonction `Factorial()` avec une condition d'erreur si la chose se produit ? Elle connaîtra l'existence du problème, mais sans savoir comment il s'est produit. Le mieux que `Factorial()` puisse faire est de signaler les erreurs à la fonction qui l'appelle (peut-être celle-ci sait-elle d'où vient le problème).

La manière classique d'indiquer une erreur dans une fonction consiste à retourner une certaine valeur que la fonction ne peut pas autrement retourner. Par exemple, la valeur d'une factorielle ne peut pas être négative. La fonction `Factorial()` peut donc retourner -1 si un nombre négatif lui est passé, -2 pour un nombre non entier, et ainsi de suite. La fonction appelante peut alors examiner la valeur retournée : si cette valeur est négative, elle sait qu'une erreur s'est produite, et la valeur exacte indique la nature de l'erreur.

Le programme `FactorialErrorReturn` suivant contient les ajustements nécessaires :




```
// FactorialErrorReturn - crée une fonction factorielle qui
//                        retourne une indication d'erreur quand
//                        quelque chose ne va pas
using System;
namespace FactorialErrorReturn
{
  // MyMathFunctions - collection de fonctions mathématiques
  //                   de ma création (pas encore grand-chose à montrer)
```

c'est le texte qui apparaît dans la zone de texte. Pour un bouton, c'est l'étiquette qui apparaît sur le bouton. Cette propriété a toujours le même sens, mais elle est interprétée selon le contexte.

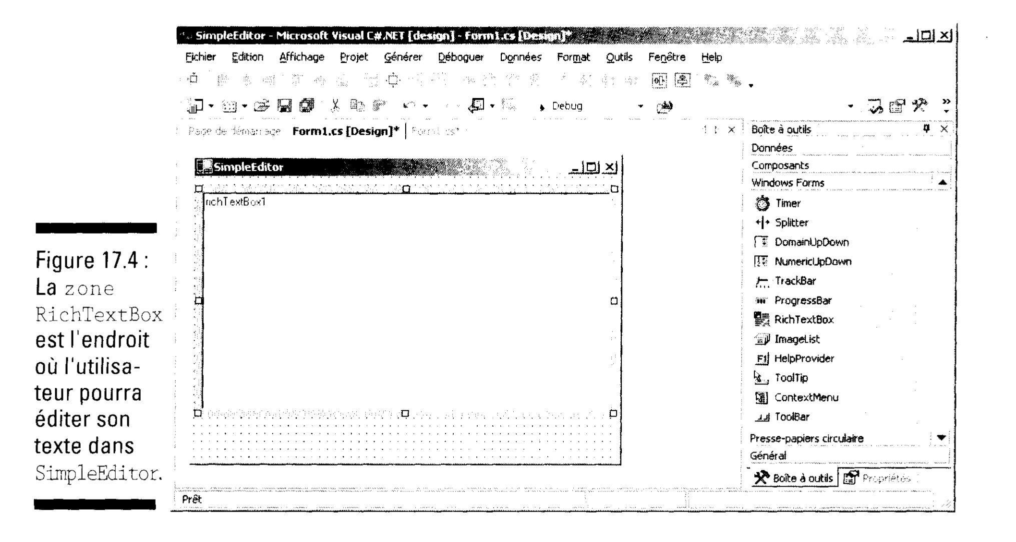


Figure 17.4 : La zone `RichTextBox` est l'endroit où l'utilisateur pourra éditer son texte dans `SimpleEditor`.

4. **Pour voir où vous en êtes arrivé, générez et exécutez à nouveau l'application.**

   `SimpleEditor` apparaît, avec la zone de texte au milieu. Vous pouvez y taper du texte, déplacer le curseur dans le but d'insérer du texte où vous voulez, et même sélectionner du texte. Bien sûr, vous ne pouvez rien faire de ce texte, mais `SimpleEditor` a déjà fait pas mal de progrès.

Construire les menus

Rien n'oblige à placer ces étapes ici, mais j'ai choisi d'ajouter maintenant les menus et leurs options. Pour cela, il vous faut un composant `MainMenu` :

1. **Dans la Boîte à outils, cliquez sur le composant** `MainMenu`**. Dans le formulaire, cliquez à l'emplacement de l'élément le plus à gauche du menu principal.**

   Un petit cadre apparaît, contenant les mots `Tapez ici`.

Traiter une erreur à l'ancienne mode : la retourner

Ne pas signaler une erreur à l'exécution n'est jamais une bonne idée. Je dis bien *jamais* : si vous n'avez pas l'intention de déboguer vos programmes et si vous ne vous souciez pas qu'ils marchent, alors seulement c'est peut-être une bonne idée.

Le programme `FactorialError` suivant montre ce qui arrive quand les erreurs ne sont pas détectées. Ce programme calcule et affiche la fonction factorielle pour de nombreuses valeurs, dont certaines sont tout juste licites.



La factorielle du nombre N est égale à N * (N-1) * (N-2) * ... * 1. Par exemple, la factorielle de 4 est 4 * 3 * 2 * 1, soit 24. La fonction factorielle n'est valide que pour les nombres entiers naturels (positifs).




```
// FactorialWithError - créer et utiliser une fonction
//                      factorielle qui ne contient aucune
//                      vérification
using System;
namespace FactorialWithError
{
  // MyMathFunctions - collection de fonctions mathématiques
  //                   de ma création (pas encore grand-chose à montrer)
  public class MyMathFunctions
  {
    // Factorial - retourne la factorielle d'une valeur
    //             fournie
    public static double Factorial(double dValue)
    {
      // commence par donner la valeur 1 à un "accumulateur"
      double dFactorial = 1.0;
      // fait une boucle à partir de nValue en descendant de 1 chaque fois
      // pour multiplier l'accumulateur
      // par la valeur obtenue
      do
      {
        dFactorial *= dValue;
        dValue -= 1.0;
      } while(dValue > 1);
      // retourne la valeur stockée dans l'accumulateur
      return dFactorial;
    }
  }
  public class Class1
  {
    public static void Main(string[] args)
```

C'est dans la méthode InitializeComponent que sont créés les composants Windows. Le commentaire spécial placé juste avant cette fonction dit en effet : "Ne touchez pas à cette section du code, parce que c'est là que moi, le Concepteur de formulaires, je fais mon boulot." En fait, le Concepteur génère le code situé entre les commentaires #region et #endregion, en réponse à ce que je dessine.

Dans ce cas simple, l'application commence par définir le membre AutoScaleBaseSize de l'objet this. Je suis pas très sûr de ce qu'est cette propriété. Heureusement, comme c'est le Concepteur de formulaires qui s'en occupe pour moi, je n'ai pas besoin de le savoir, mais je sais que this est l'objet Form lui-même. En continuant jusqu'à la dernière ligne de InitializeComponent, je peux voir que "Simple Editor" est assigné à this.Text.

Prenez le temps d'étudier soigneusement ce point, car c'est là l'essentiel du Concepteur de formulaires. Le Concepteur affiche les propriétés du formulaire. L'une de ces propriétés est Text, à laquelle j'ai donné la valeur "Simple Editor". Le Concepteur a ajouté une ligne de code qui assigne en conséquence cette valeur à la propriété Text du formulaire.

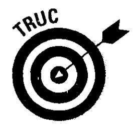
La méthode Dispose est invoquée lorsque Form1 est fermé. Elle n'est pas particulièrement intéressante dans ce cas, parce que la fermeture du formulaire ferme aussi l'éditeur.

Éditer la fenêtre d'édition

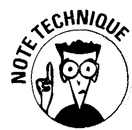
La propriété la plus importante de SimpleEditor est la fenêtre d'édition :

1. **Ouvrez la Boîte à outils en sélectionnant Affichage/Boîte à outils.**

   La Boîte à outils contient une collection d'objets graphiques en C#, que l'on appelle aussi parfois *composants*. On y trouve divers ensembles d'accessoires, dont un ensemble nommé Windows Forms, contenant les accessoires dont nous avons besoin pour réaliser SimpleEditor.



Le terme *composant* ne s'applique pas seulement aux objets graphiques, mais tous les objets graphiques sont des composants. C'est donc ce terme que nous utilisons ici.



Les accessoires de données sont utilisés pour réaliser facilement des liens avec des bases de données externes. Les composants gèrent le multitâche. La section Général de la Boîte à outils est l'endroit où vous pouvez stocker les accessoires que vous réalisez

C# résout ce problème en allouant tous les objets à partir du tas. Mieux encore, C# retourne la mémoire au tas pour vous. Plus d'écran noir parce que vous avez envoyé au tas le mauvais bloc de mémoire.

Les variables de type pointeur ne sont pas autorisées

L'introduction des pointeurs par le langage C a beaucoup fait pour son succès. Les manipulations de pointeur étaient une fonctionnalité puissante. Les vétérans de la programmation en langage machine pouvaient y reproduire les astuces de programmation qui leur étaient familières. C++ a conservé de C, sans modifications, les fonctionnalités sur les pointeurs et le tas.

Malheureusement, ni le programmeur ni le programme ne peuvent distinguer un bon pointeur d'un mauvais. Lisez un bloc de mémoire avec un pointeur non initialisé et, si vous avez de la chance, votre programme se plante. Si vous n'avez pas de chance, le programme poursuit son petit bonhomme de chemin en traitant comme un objet valide le bloc de mémoire trouvé au hasard.

Les problèmes de pointeur sont souvent difficiles à identifier. Un programme qui contient un pointeur invalide se comporte en général de façon différente à chaque exécution.

Heureusement pour tous ceux qui sont concernés, C# a écarté les problèmes de pointeur en se débarrassant des pointeurs en général. Les références qu'il utilise à la place sont indépendantes du type et ne peuvent pas être manipulées par l'utilisateur pour en faire quelque chose qui pourrait démolir le programme.

Vendez-moi quelques-unes de vos propriétés

Tout bon programmeur sait que l'accès à un membre donnée doit être soigneusement contrôlé à l'aide d'une méthode `get()` pour en retourner la valeur, et éventuellement d'une méthode `set()` pour lui assigner une valeur. Tout programmeur qui a déjà utilisé les fonctions `get()` et `set()` est conscient du fait que leur emploi n'est pas une chose très naturelle :

```
using System;
public class Student
```

fichiers "include", qui sont alors utilisés par les modules ; toutefois, il peut devenir très compliqué de placer tous ces fichiers include dans le bon ordre pour que votre module se compile correctement.

C# se débarrasse de cette absurdité en recherchant lui-même les définitions de classe, et en les trouvant. Si vous invoquez une classe Student, C# recherche et trouve lui-même la définition de cette classe pour s'assurer que vous l'utilisez correctement. Il n'a pas besoin pour cela que vous lui donniez un indice.

Ne construisez pas, initialisez

J'ai trouvé évidente l'utilité des constructeurs la première fois que j'ai jeté les yeux sur l'un d'eux. Fournir une fonction spéciale pour s'assurer que tous les membres donnée ont été définis correctement ? Quelle bonne idée ! Le seul inconvénient, c'est que j'ai fini par ajouter un constructeur trivial chaque fois que j'écrivais une classe :

```
public class Account
{
  private double balance;
  private int numChecksProcessed;
  private CheckBook checkBook;
  public Account()
  {
    balance = 0.0;
    numChecksProcessed = 0;
    checkBook = new CheckBook();
  }
}
```

Pourquoi ne pourrais-je pas initialiser directement un membre donnée en laissant le langage générer le constructeur pour moi ? C++ demande pourquoi, C# répond pourquoi pas ? C# se débarrasse des constructeurs inutiles en autorisant l'initialisation directe :

```
public class Account
{
  private double balance = 0.0;
  private int numChecksProcessed = 0;
  private CheckBook checkBook = new CheckBook();
}
```

Les programmeurs ont ensuite réalisé que l'interface, plus légère, pouvait rendre les mêmes services. Une classe qui implémente une interface, comme l'exemple suivant, promet à C# comme à tout le monde qu'elle offre les méthodes `read()` et `write()` correspondantes :

```
interface IPersistable
{
  void read();
  void write();
}
```

Le système des types unifiés

En C++, la classe est une fonctionnalité fort sympathique. Elle permet aux données et à leurs fonctions associées d'être rassemblées dans un ensemble propre et net, fait pour reproduire la manière dont les gens voient les choses dans le monde réel. Le seul inconvénient est que tout langage doit offrir de la place pour les types de variable simples comme les entiers et les nombres en virgule flottante. Cette nécessité a produit un système de castes. Les objets de classe vivaient d'un côté, et les variables de type valeur comme `int` et `float` vivaient de l'autre. Bien sûr, les types valeur et les types objet étaient autorisés à jouer dans le même programme, mais le programmeur devait maintenir cette séparation dans son esprit.

C# abat le mur de Berlin qui séparait les types valeur des types objet. Pour chaque type valeur, il y a une "classe de type valeur" correspondante, que l'on appelle une *structure*. Ces structures à faible coût peuvent se mélanger librement avec les objets de classe, permettant aux programmeurs d'écrire des instructions comme celles-ci :

```
MyClass myObject = new MyClass();
// affiche un "myObject" mis sous forme de chaîne
Console.WriteLine(myObject.ToString());
int i = 5;
// affiche un int sous forme de chaîne
Console.WriteLine(i.ToString());
// affiche la constante 5 sous forme de chaîne
Console.WriteLine(5.ToString());
```

Non seulement je peux invoquer la même méthode sur un `int` que sur un objet de `MyClass`, mais je peux aussi le faire avec une constante comme "5". Ce scandaleux mélange des types de variable est une fonctionnalité puissante de C#.

Index

D

E

J – K

M

N

Achevé d'imprimer
en janvier 2002
par Legoprint S.p.A.
Lavis, Italie

O

P

Q – R

Définis soigneusement tes types de variable, mon enfant

C++ est très politiquement correct. Il ne marcherait à aucun prix sur les plates-bandes d'un ordinateur en exigeant qu'un type de variable particulier soit limité à une étendue de valeurs particulière. Il spécifie qu'un `int` fait à peu près "telle taille" et qu'un long est "plus grand". Cette décision conduit à des erreurs obscures quand on essaie de déplacer un programme d'un type de processeur à un autre.

C# n'y va pas par quatre chemins. Il dit qu'un `int` fait 32 bits et qu'un long fait 64 bits, et que c'est comme ça. En tant que programmeur, vous pouvez envoyer cela à votre banque ou à un autre ordinateur sans qu'il en résulte d'erreurs inattendues.

Pas d'héritage multiple

C++ autorise une même classe à hériter de plus d'une classe de base. Par exemple, une classe `CanapéLit` peut hériter de la classe `Lit` et de la classe `Canapé`. Ça ne semble pas manquer de rigueur, et ça peut effectivement être très utile. Le seul inconvénient, c'est que l'héritage de plusieurs classes de base peut provoquer des erreurs de programmation qui sont parmi les plus difficiles à identifier que l'on connaisse.

C# se met en retrait, et évite des erreurs supplémentaires en écartant l'héritage multiple. Toutefois, ce choix n'aurait pas été possible si C# n'avait pas remplacé l'héritage multiple par une nouvelle fonctionnalité : l'interface.

Prévoir une bonne interface

Quand les gens ont pris un peu de recul pour réaliser dans quel cauchemar ils s'étaient mis avec l'héritage multiple, ils se sont rendu compte que dans 90 % des cas, la deuxième classe de base n'était là que pour décrire la sous-classe. Par exemple, une classe parfaitement ordinaire pouvait hériter d'une classe abstraite `Persistable`, avec une méthode abstraite `read()` et une autre `write()`. Cela obligeait la sous-classe à implémenter les méthodes `read()` et `write()` et à dire au monde extérieur que ces méthodes étaient disponibles si on voulait s'en servir.

```
{
  private string sName;
  public void set(string sName)
  {
    this.sName = sName;
  }
  public string get()
  {
    // retourne une copie du nom
    return String.Copy(sName);
  }
}
class MyClass
{
  public void AddLastName(Student student)
  {
    student.set(student.get() + " Kringle");
  }
}
```

La notion de Propriété en C# permet d'implémenter les fonctions `get()` et `set()` d'une manière complètement naturelle dans le programme :

```
using System;
public class Student
{
  private string sName;
  public string Name
  {
    set { sName = value;}
    get { return String.Copy(sName); }
  }
}
class MyClass
{
  public void AddLastName(Student student)
  {
    student.Name = student.Name + "Kringle";
  }
}
```

Je n'inclurai plus jamais un fichier

C++ impose une cohérence rigoureuse des types. C'est une bonne chose. Il le fait en vous obligeant à déclarer vos fonctions et vos classes dans des

Pas de données ni de fonctions globales

C++ passe pour un langage orienté objet, et il l'est, au sens où vous pouvez l'utiliser pour programmer d'une manière orientée objet. Vous pouvez aussi mettre de côté les objets en plaçant simplement les données et les fonctions dans un espace global, ouverts à tous les éléments et à tout programmeur doté d'un clavier.

C# demande au programmeur de lui faire allégeance : toutes les fonctions et tous les membres donnée doivent faire partie d'une classe. Si vous voulez accéder à cette fonction ou à ces données, vous devez passer par l'auteur de cette classe. Il n'y a pas d'exception à cela.

Tous les objets sont alloués à partir du tas

C comme C++ autorisent l'allocation de mémoire de trois manières différentes, chacune avec ses propres inconvénients :

- **Les objets globaux existent du début à la fin de l'exécution du programme.** Un programme peut facilement allouer plusieurs pointeurs au même objet global. Si vous en modifiez un, ils sont tous modifiés, qu'ils soient prêts pour cela ou non.
- **Un objet de pile est propre à une fonction (ce qui est une bonne chose), mais son allocation disparaît lorsque la fonction retourne son résultat.** Tout pointeur qui pointe vers un objet dont l'allocation mémoire a été supprimée devient invalide. Ce serait très bien si quelqu'un avait prévenu le pointeur, mais le malheureux s'imagine toujours qu'il pointe vers un objet valide, et le programmeur aussi.
- **Les objets du tas sont alloués en fonction des nécessités.** Ces objets sont propres à un thread d'exécution particulier.

Le problème est qu'il est trop facile d'oublier à quel type de mémoire se réfère un pointeur. Un objet du tas doit être retourné quand on en a fini avec lui. Oubliez-le, et votre programme aura de moins en moins de mémoire disponible, jusqu'à ce qu'il ne puisse plus fonctionner. D'un autre côté, si vous libérez plus d'une fois le même bloc du tas pour "retourner" un bloc de la mémoire globale ou de la pile, votre programme est parti pour une longue sieste. Il faudra peut-être Ctrl+Alt+Del pour le réveiller.

Ces deux possibilités sont mises en évidence dans la classe suivante :

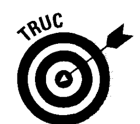
```
public class MyClass
{
  public string ConvertToString(int n)
  {
    // convert the int n into a string s
    string s = n.ToString();
  }
  public string ConvertPositiveNumbers(int n)
  {
    // only positive numbers are valid for conversion
    if (n > 0)
    {
      string s = n.ToString();
      return s;
    }
              Console.WriteLine("the argument {0} is invalid", n);
  }
}
```

`ConvertToString()` calcule une chaîne à retourner, mais ne la retourne jamais. Ajoutez simplement `return s;` en bas de la méthode.

`ConvertPositiveNumbers()` retourne la version chaîne de l'argument `int n` lorsque celui-ci est positif. En outre, il génère correctement un message d'erreur lorsque `n` n'est pas positif. Mais même si `n` n'est pas positif, la fonction doit retourner *quelque chose*. Dans ces cas-là, retournez soit un `null`, soit une chaîne vide "". La solution qui conviendra le mieux dépend de l'application.

} attendue

Ce message indique que C# attendait encore une accolade fermante à la fin du listing. Quelque part dans le code, vous avez oublié de fermer une définition de classe, une fonction ou un bloc `if`. Reprenez votre code et appariez soigneusement les accolades ouvrantes et fermantes jusqu'à ce que vous trouviez la coupable.



Ce message est souvent le dernier d'une série de messages d'erreur souvent idiots. Ne vous préoccupez pas des autres avant d'avoir remédié à celui-ci.

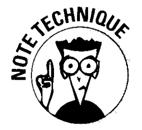
```
    new public void Function()
    {
    }
  }
```

- Vous vouliez vraiment hériter de la classe de base par polymorphisme, auquel cas vous auriez dû déclarer les deux classes de la façon suivante :

```
public class BaseClass
{
  public virtual void Function()
  {
  }
}

public class SubClass : BaseClass
{
  public overrides void Function()
  {
  }
}
```

Voyez le Chapitre 13 pour en savoir plus.



Cela n'est pas une erreur, mais seulement un avertissement dans la fenêtre Liste des tâches.

'subclassName' : ne peut pas hériter de la classe scellée 'baseclassName'

Ce message indique que la classe est scellée et que vous ne pouvez donc pas en hériter, ni en modifier les propriétés. Typiquement, seules les classes des bibliothèques sont scellées. Vous ne pouvez rien y changer.

'className' n'implémente pas le membre d'interface 'methodName'

L'implémentation d'une interface représente une promesse de fournir une définition pour toutes les méthodes que comporte cette interface. Ce message vous dit que vous n'avez pas tenu cette promesse car vous n'avez pas implémenté la méthode citée. Il peut y avoir plusieurs raisons à cela :

```
        n = 1;
    }
}
```

Dans ce cas, aucune valeur n'est assignée à n dans SomeFunction(), mais elle en reçoit une dans SomeOtherFunction(). Cette dernière ignore la valeur d'un argument out comme s'il n'existait pas, ce qui est le cas ici.

Le fichier 'programName.exe' ne peut pas être copié dans le répertoire d'exécution. Le processus ne peut pas...

En général, ce message se répète de nombreuses fois. Dans presque tous les cas, il signifie que vous avez oublié d'arrêter le programme avant de le générer à nouveau. Autrement dit, voilà ce que vous avez fait :

1. **Vous avez généré votre programme avec succès (supposons que ce soit une application console, bien que cela puisse se produire avec n'importe quelle sortie en C#).**

2. **Vous avez vu le message "Appuyez sur Entrée pour terminer", mais, dans votre hâte, vous ne l'avez pas fait. Votre programme est donc toujours en cours d'exécution, et vous êtes retourné dans Visual Studio pour modifier le fichier.**

3. **Vous avez essayé de générer à nouveau le programme avec vos modifications. C'est là que vous avez obtenu ce message d'erreur.**

Un fichier exécutable .EXE est verrouillé par Windows jusqu'à ce que le programme termine effectivement son exécution. Visual C# ne peut pas écraser la version précédente du fichier exécutable .EXE avec la nouvelle version tant que le programme n'a pas terminé son exécution.

Revenez à l'application, et faites le nécessaire pour qu'elle se termine. Dans le cas d'une application console, appuyez simplement sur la touche Entrée. Vous pouvez aussi mettre fin à l'exécution d'un programme dans Visual Studio en sélectionnant Déboguer/Arrêter le déboguage.

Une fois que la version antérieure du programme a terminé son exécution, générez à nouveau l'application.

Si vous n'arrivez pas à vous débarrasser de l'erreur en mettant fin à l'exécution du programme, il est possible qu'il y ait quelque chose qui

Le résultat de 2.0 * f est toujours de type double, mais le programmeur a indiqué qu'il voulait que le résultat soit converti en type float, même au cas improbable où il en résulterait une perte d'informations.

Une autre approche consisterait à s'assurer que toutes les constantes sont de même type :

```
class MyClass
{
  static public float FloatTimes2(float f)
  {
    // ceci fonctionne bien parce que 2.0F est une constante de type float
    float fResult = 2.0F * f;
    return fResult;
  }
}
```

Cette version de la fonction utilise une constante 2.0 de type float au lieu du type double par défaut. Un float multiplié par un float est un float.

'className.memberName' est inaccessible en raison de son niveau de protection

Ce message indique qu'une fonction essaie d'accéder à un membre auquel elle n'a pas accès. Par exemple, une méthode d'une classe peut essayer d'accéder à un membre privé d'une autre classe (voyez le Chapitre 11) :

```
public class MyClass
{
  public void SomeFunction()
  {
    YourClass uc = new YourClass();
    // ceci ne fonctionne pas correctement parce que MyClass
    // ne peut pas accéder au membre privé
    uc.nPrivateMember = 1;
  }
}
public class YourClass
{
  private int nPrivateMember = 0;
}
```

En général, l'erreur n'est pas aussi flagrante. Bien souvent, vous avez simplement laissé le descripteur hors de l'objet membre ou de la classe elle-même.

Impossible de convertir implicitement le type 'x' en 'y'

Ce message indique généralement que vous essayez d'utiliser deux types de variable différents dans la même expression. Par exemple :

```
int nAge = 10;
// génère un message d'erreur
int nFactoredAge = 2.0 * nAge;
```

Le problème est ici que 2.0 est une variable de type `double`. La variable `nAge` de type `int` multipliée par le 2.0 de type `double` produit une valeur de type `double`. C# ne va pas automatiquement stocker une valeur de type `double` dans la variable de type `int nFactoredAge`, car il pourrait en résulter une perte d'information (en particulier, la partie décimale de la valeur `double`).

Certaines conversions ne sont pas aussi évidentes, comme dans l'exemple suivant :

```
class MyClass
{
  static public float FloatTimes2(float f)
  {
    // ceci produit une erreur de génération
    float fResult = 2.0 * f;
    return fResult;
  }
}
```

On pourrait croire que multiplier par deux un type `float` ne pose pas de problème, mais c'est justement là qu'est le problème. 2.0 n'est pas de type `float` mais de type `double`. Un `float` multiplié par un `double` donne un `double`. C# ne va pas stocker une valeur de type `double` dans une variable de type `float`, à cause – vous avez deviné – de la perte d'informations qui pourrait en résulter (dans ce cas, plusieurs chiffres de précision).

Les conversions implicites peuvent déconcerter plus encore le lecteur désinvolte (c'est mon cas, dans les bons jours). Cette version de `FloatTimes2()` marche très bien :

```
class MyClass
{
  static public float FloatTimes2(float f)
  {
```

parfois affreusement bavard. J'ai réduit certains des messages d'erreur pour les faire tenir sur une page. En plus, il y a dans un message d'erreur différents endroits où apparaît le nom d'un membre donnée offensant ou d'une classe irrévérencieuse. J'ai remplacé ces noms par `variableName`, `memberName` ou `className`.

Enfin, C# ne se contente pas de cracher le nom de la classe. Il préfère mettre sur la table l'espace de nom au grand complet (au cas, bien sûr, où le message aurait été trop court avec la première solution).

'className' ne contient pas de définition pour 'memberName'

Ce message d'erreur peut signifier que vous avez oublié de déclarer une variable, comme dans l'exemple suivant :

```
for(index = 0; index < 10; index++)
{
  // . . . instructions . . .
}
```

La variable index n'est définie nulle part (pour savoir comment déclarer les variables, reportez-vous au Chapitre 3). Cet exemple devrait avoir été écrit de la façon suivante :

```
for(int index = 0; index < 10; index++)
{
  // . . . instructions . . .
}
```

La même chose s'applique aux membres donnée d'une classe (voyez le Chapitre 6).

Il y a en fait plus de chances que vous ayez fait une faute de frappe dans un nom de variable. En voici un bon exemple :

```
class Student
{
  public string sStudentName;
  public int nID;
}
class MyClass
{
```

Dans cette partie...

Quel livre *Pour les nuls* serait complet sans notre traditionnelle partie des dix ? C# est très doué pour trouver des erreurs dans vos programmes lorsque vous essayez de les générer. Vous l'avez sans doute remarqué. Mais les messages d'erreur qu'il génère peuvent être assez obscurs. Vous l'avez sans doute remarqué aussi. Le Chapitre 19 passe en revue les dix messages d'erreur de générations les plus courants et ce qu'ils signifient le plus souvent. Et comme savoir c'est pouvoir, vous y trouverez aussi des suggestions de correction pour les problèmes correspondants.

Beaucoup des lecteurs de ce livre seront venus à C# par le plus répandu de tous les langages orientés objet, C++. Le Chapitre 20 donne la liste des dix différences principales entre ces deux langages.

Implémenter le bouton de fermeture de la fenêtre

Il reste encore un petit problème. Il est toujours possible de quitter l'application en fermant la fenêtre :

1. **Dans le Concepteur de formulaires, sélectionnez le cadre de la fenêtre du programme.**

2. **Dans la fenêtre Propriétés, sélectionnez l'événement** `Closing`**.**

3. **Entrez le nom de fonction** ApplicationWindowClosing**.**

   C'est la propriété `Closing` qui est invoquée lorsque l'utilisateur clique sur le bouton de fermeture de la fenêtre (le x dans le coin supérieur droit). Il est facile d'associer une méthode à cette propriété. La difficulté est de savoir quoi faire quand elle reçoit le contrôle. La réponse est donnée par un aspect des méthodes de propriété, que j'ai passé sous silence jusqu'ici.

   Lorsque C# appelle une méthode en réponse à un clic sur un bouton ou à la saisie d'une valeur, il lui passe deux arguments. Le premier de ceux-ci est appelé le `sender`. Cet objet est le composant qui est à l'origine du stimulus. Les méthodes que nous avons générées ne pouvaient être invoquées que par une seule source. Toutefois, il peut être utile de différencier les senders, car cela réduit la quantité de code à écrire (je suis toujours d'accord pour écrire moins de code). Par exemple, une même méthode peut être utilisée pour traiter plusieurs boutons radio, le sender indiquant quel est le bouton sur lequel a cliqué l'utilisateur.

   Le deuxième argument contient d'autres informations sur l'événement, qui ont trait à la raison pour laquelle la méthode a été appelée. Toutefois, la classe `CancelEventArgs` passée à notre méthode à la suite d'un clic sur le bouton de fermeture de la fenêtre contient une propriété `Cancel` (Annuler). Si vous donnez à cet indicateur la valeur `true`, l'opération de fermeture de la fenêtre est tuée dans l'œuf.

4. **Modifiez la méthode** `ApplicationWindowClosing()` **sur la base de la même logique "ne pas perdre les modifications" que nous avons utilisée avec succès pour la commande Fichier/Quitter :**

MessageBox retourne DialogResult.Yes, c'est que l'utilisateur dit qu'il est d'accord pour ne pas enregistrer ses modifications.

C'est ce que réalisent les modifications suivantes aux méthodes FileOpen(), FileSave() et FileExit() :

```
private void FileOpen(object sender, System.EventArgs e)
{
  // n'écrase pas le précédent fichier en ouvrant le nouveau
  if (IsChangeOK() == false)
  {
    return;
  }
  // tout va bien, dit-il
  OpenAndReadFile();
  // la zone de texte est maintenant dans l'état non modifié
  bTextChanged = false;
}
private void FileSave(object sender, System.EventArgs e)
{
  // donne la valeur false à l'indicateur de modification
  // si l'enregistrement a fonctionné
  if (SaveSpecifiedFile())
  {
    bTextChanged = false;
  }
}
private void FileExit(object sender, System.EventArgs e)
{
  // ne quitte pas si les modifications n'ont pas été déjà enregistrées
  // ou tant que l'utilisateur ne dit pas qu'il est d'accord
  if (IsChangeOK() == false)
  {
    return;
  }
  // la voie est libre
  Application.Exit();
}
```

Dans tous les cas, la méthode IsChangeOK() est invoquée avant d'exécuter une opération qui pourrait provoquer la perte d'une modification.

Il nous manque encore quelque chose : il faut que l'indicateur bTextChanged reçoive la valeur true à chaque modification apportée dans la zone de texte. RichTextBox nous donne ce qu'il nous faut : la propriété dynamique TextChanged(), qui est invoquée chaque fois que quelque chose modifie le

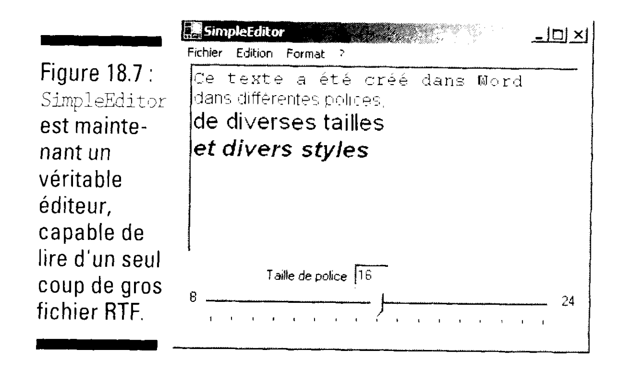


Figure 18.7 : SimpleEditor est maintenant un véritable éditeur, capable de lire d'un seul coup de gros fichier RTF.

Ne perdez pas mes modifications en quittant !

L'implémentation de la commande Fichier/Quitter est une chose facile :

1. **Dans le Concepteur de formulaires, sélectionnez l'option de menu Fichier/Quitter.**

2. **Dans la fenêtre Propriétés, sélectionnez l'événement** `Click`**, et entrez le nom de fonction** FileExit.

3. **Implémentez de la façon suivante la méthode** `FileExit()` **que vous venez de créer :**

```
private void FileExit(object sender, System.EventArgs e)
{
  Application.Exit();
}
```

C'est la classe `Application` qui contrôle l'ensemble du programme. Comme son nom l'indique, sa méthode `Exit()` fait directement sortir de la scène. L'inconvénient de cette solution est que quitter le programme sans avoir commencé par faire Fichier/Enregistrer provoque la perte de vos dernières modifications. Ce n'est pas une chose à faire.

Heureusement, les méthodes de lecture et d'écriture nous donnent ce dont nous avons besoin pour éviter cette catastrophe. `SaveSpecifiedFile()` retourne un `bool` qui indique si la donnée a effectivement été enregistrée. Nous n'avons besoin ici que d'un indicateur "grossier", disant s'il y a ou non dans `RichTextBox` quelque chose à enregistrer. Nous pouvons assigner `false`

```
      if (strOutput != null)
      {
        System.IO.StreamWriter strWtr =
                      new System.IO.StreamWriter(strOutput);
        strWtr.Write(richTextBox1.Rtf);
        strWtr.Close();
        bReturnValue = true;
      }
    }
    return bReturnValue;
  }
```

Cette fonction suit exactement le même chemin que la méthode OpenAndReadFile() que nous avons vue plus haut. Pour commencer, elle ouvre la boîte de dialogue SaveFileDialog, et attend qu'elle lui retourne OK. Le programme essaie alors d'ouvrir le fichier. Si l'opération aboutit, le programme écrit dans le fichier tout le contenu de la propriété Rtf de la zone de texte RichTextBox.



Nous avons ajouté le composant SaveFileDialog à SimpleEditor en le faisant glisser depuis la Boîte à outils. Aucun ajustement n'a été nécessaire.

Mettre Lire et Écrire dans une boîte, avec un menu par-dessus

Les méthodes OpenAndReadFile() et SaveSpecifiedFile() sont bien jolies, mais totalement inutiles tant qu'elles ne sont pas liées chacune à une option de menu.

Pour implémenter les options de menu Fichier/Ouvrir et Fichier/Enregistrer, suivez ces étapes :

1. **Entrez les méthodes** OpenAndReadFile() **et** SaveSpecifiedFile() **décrites dans les sections précédentes.**

2. **Dans le Concepteur de formulaires, sélectionnez l'option de menu Fichier/Enregistrer.**

3. **Dans la fenêtre Propriétés, sélectionnez l'événement** Click**, puis entrez le nom de fonction** FileSave**, comme le montre la Figure 18.6.**

Lire un fichier RTF

La boîte de dialogue OpenFileDialog est étonnamment facile à utiliser. La méthode ShowDialog() ouvre la boîte de dialogue. SimpleEditor n'a rien à faire pendant que l'utilisateur fait défiler le contenu de cette boîte de dialogue à la recherche du fichier à ouvrir. Une fois qu'il a terminé, il clique sur OK (ou sur Annuler s'il n'a rien trouvé). C'est seulement alors que le contrôle est restitué à la fonction appelante. La valeur retournée par ShowDialog() est DialogResult.OK si l'utilisateur a cliqué sur le bouton OK. S'il a cliqué sur autre chose, ça ne nous intéresse pas.

La méthode OpenFile() retourne soit un IO.Stream valide qui permet de lire le fichier, soit un null si le fichier spécifié ne peut pas être lu pour une raison ou pour une autre. Ces deux méthodes sont combinées dans la fonction OpenAndReadFile() suivante :

```
// lit dans la RichTextBox le fichier spécifié par l'utilisateur
// (retourne true si la RichTextBox est modifiée)
private bool OpenAndReadFile()
{
  bool bReturnValue = false;
  try
  {
    // lit le nom de fichier entré par l'utilisateur
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
      // ouvre le fichier
      System.IO.Stream strInput = openFileDialog1.OpenFile();
      if (strInput != null)
      {
        // si l'ouverture du fichier a réussi, lui associe
        // un lecteur de flux
        System.IO.StreamReader strRdr =
                    new System.IO.StreamReader(strInput);
        // lit tout le contenu du fichier
        string sContents = strRdr.ReadToEnd();
        richTextBox1.Rtf = sContents;
        // nous avons modifié la fenêtre de texte
        bReturnValue = true;
        // assurons-nous de fermer le fichier pour que d'autres
        // puissent le lire
        strRdr.Close();
      }
    }
  }
    catch(Exception e)
```

4. **Générez le programme et exécutez-le.**
5. **Entrez du texte et sélectionnez-le.**
6. **Entrez une taille de police entre 8 et 24.**

   La taille du texte sélectionné change, et l'index de la barre se déplace à la position correspondante.

Enregistrer le texte de l'utilisateur

Sans la possibilité de lire et d'écrire des fichiers, `SimpleEditor` ne serait rien de plus qu'un jouet.

Lire le nom du fichier

Pour lire un fichier, il faut savoir lequel il faut lire. Pour cela, C# fournit une boîte de dialogue spéciale, nommée `OpenFileDialog`. Tôt ou tard, l'utilisateur voudra enregistrer sur le disque le texte qu'il vient de saisir et de mettre en forme. Vous avez deviné : vous avez besoin pour cela de `OpenFileDialog`. Associez à ces deux boîtes de dialogue les fonctions de lecture et d'écriture de fichier que nous avons vues au Chapitre 16, et vous avez un éditeur complet.

Ajouter une boîte de dialogue `OpenFileDialog` est un jeu d'enfant :

1. **Dans le Concepteur de formulaires, faites glisser un composant** `OpenFileDialog` **de la Boîte à outils jusqu'à la zone qui se trouve au-dessous de la fenêtre d'édition (la zone dans laquelle il y a déjà MainMenu1).**
2. **Faites la même chose avec un composant** `SaveFileDialog`**.**

   Le résultat doit ressembler à la Figure 18.5.

   Le composant `OpenFileDialog` ne contient qu'une seule propriété statique vraiment intéressante. Quand on utilise cette boîte de dialogue pour ouvrir un fichier, il y apparaît généralement dans une liste des fichiers de divers types. Par exemple, le Bloc-notes commence par rechercher les fichiers *.txt, alors que Word commence par rechercher les fichiers *.doc. C'est ce qu'on

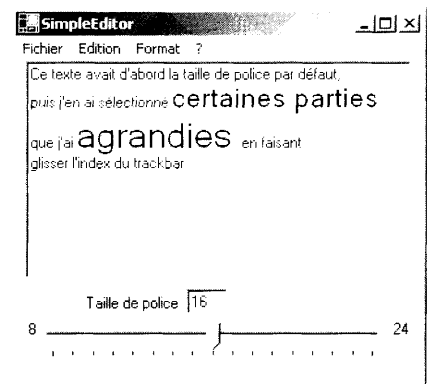


Figure 18.4 : Grâce au lien établi entre ces deux composants, la valeur qui apparaît dans la `TextBox` est mise à jour en fonction de la position de l'index dans la `TrackBar`.

Changer de taille en utilisant la `TextBox`

L'utilisateur peut aussi entrer directement une taille de police dans la zone de texte Taille de police (sinon, elle ne servirait pas à grand-chose). Cette fonction doit donc fonctionner dans la direction opposée à la fonction `FontSizeControl()`, mais elle est également un peu plus compliquée car elle doit prendre en compte les erreurs de saisie éventuelles de l'utilisateur. Mais au bout du compte, `FontSizeEntered()` doit exécuter la même opération : lire la nouvelle valeur, modifier la taille de police, et ajuster la position de l'index de la barre en conséquence.

1. **Sélectionnez l'objet `TextBox` de la taille de police.**

2. **Dans la fenêtre Propriétés, sélectionnez l'événement `TextChanged`, et entrez le nom de fonction** FontSizeEntered.

   C'est cette méthode qui sera invoquée par C# lorsque l'utilisateur entrera une nouvelle valeur dans la zone de texte Taille de police.

3. **Dans le code source, implémentez la nouvelle fonction comme suit :**

```
// invoquée quand l'utilisateur tape quelque chose dans la TextBox
// utilisé pour définir la taille de la police
private void FontSizeEntered(object sender, System.EventArgs e)
{
  // lit le contenu de la TextBox
  string sText = textBox1.Text;
```

```
    //
    // TODO: Add any constructor code after InitializeComponent call
    //
    // donne à la police le style et la taille par défaut
    SetFont();
}
```

6. **Générez le programme et exécutez-le.**

7. **Entrez du texte, et sélectionnez-le avec la souris.**

8. **Sélectionnez Format/Gras et Format/Italique dans un ordre quelconque.**

La Figure 18.3 montre du texte mis en gras et en italique dans la fenêtre de `SimpleEditor`. Remarquez aussi que les options activées à l'endroit où se trouve le curseur sont précédées d'une coche dans le menu, qui vous permet donc de savoir quelle sera la mise en forme du texte que vous allez taper à partir de là. C'est le résultat de notre définition de la propriété `Checked`.

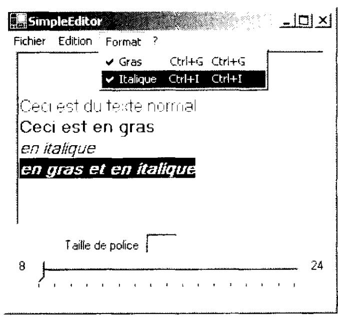


Figure 18.3 : `SimpleEditor` permet maintenant de saisir du texte en gras et en italique.

Choisir la taille de police

Le changement de la taille de police utilise la même fonction `SetFont()`, mais avec une petite complication, car elle peut être définie par deux composants différents.

Changer de taille en utilisant la `TrackBar`

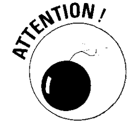
C'est une opération qui se fait assez directement :

```
    fs |= FontStyle.Bold;
  }
  if (isItalics)
  {
    fs |= FontStyle.Italic;
  }
  Font font = new Font(richTextBox1.Font.FontFamily, fontSize, fs);
  richTextBox1.SelectionFont = font;
}
```

La base de cette fonction est le constructeur `Font()`. Il en existe de nombreuses versions, mais celle-ci admet les arguments qui nous intéressent : la police courante, la nouvelle taille, et la nouvelle police. `FontStyle` rassemble sous forme de bits des propriétés comme gras, italique, barré et souligné. Commencez par `FontStyle.Regular` et ajoutez celles que vous voulez de ces propriétés en utilisant l'opérateur C# OR (| |). Les deux indicateurs `isBolded` et `isItalics` stockent les informations disant si le texte est ou non en gras ou en italique.

Le premier argument du constructeur spécifie la police courante (la possibilité de changer de police ne fait pas partie des fonctionnalités que nous avons retenues pour `SimpleEditor`). La commande `richTextBox1.Font` retourne une description de la police courante. La propriété `FontFamily` retourne le type de police (par exemple, "Arial" ou "Times New Roman"). Le constructeur crée donc un nouvel objet `Font`, avec la même police mais dans une nouvelle taille, et dont les attributs gras et italiques ont éventuellement été changés.

La dernière assignation modifie la police du texte sélectionné ou du texte qui sera tapé à partir du point où se trouve le curseur.



L'expression `richTextBox1.Font = font;` modifie la police de tout le texte qui se trouve dans la zone de texte.

Implémenter les options du menu Format

Les étapes ci-dessous implémentent les options du menu Format :

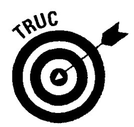
1. **Dans les menus, sélectionnez Format/Gras.**
2. **Dans la fenêtre Propriétés, sélectionnez l'événement** `Click`**, et entrez le nom** FormatBold.
3. **Répétez le même processus pour l'option de menu Format/ Italique, en utilisant le nom** FormatItalics.

```
    // efface ce qui est déjà là
    richTextBox1.SelectedRtf = "";
  }

  private void EditCopy(object sender, System.EventArgs e)
  {
    string rtfText = richTextBox1.SelectedRtf;
    WriteClipboard(rtfText);
  }

  private void EditPaste(object sender, System.EventArgs e)
  {
    string s = ReadClipboard();
    if (s != null)
    {
      richTextBox1.SelectedRtf = s;
    }
  }
```

La propriété `SelectedRtf` contient à chaque instant le texte qui est sélectionné. La méthode `EditCopy()` passe cette propriété à `WriteClipboard()`. La méthode `EditCut()` fait la même chose, mais en supprimant le texte sélectionné en assignant une chaîne vide à cette propriété. La méthode `EditPaste()` lit dans le Presse-papiers une chaîne RTF, par laquelle il remplace le texte sélectionné (ou insère cette chaîne à l'endroit où se trouve le curseur s'il n'y a pas de texte sélectionné).



Double-cliquer sur le nom d'une propriété dans la fenêtre Propriétés vous conduit directement à la fonction correspondante. Cette astuce peut faire gagner pas mal de temps.

8. **Générez à nouveau `SimpleEditor`. Vous avez maintenant un programme qui peut vraiment couper, copier et coller.**

9. **Exécutez le programme en sélectionnant Déboguer/Démarrer.**

10. **Tapez quelques lignes de texte dans la fenêtre d'édition.**

11. **Sélectionnez une portion de texte, sélectionnez Édition/Couper (ou appuyez sur Ctrl+X), placez le curseur à l'endroit que vous voulez, sélectionnez Édition/Coller (ou appuyez sur Ctrl+V), et voilà ! Le texte a été déplacé.**

Plus impressionnant encore, `SimpleEditor` peut échanger du texte par Couper et Coller avec d'autres applications, par exemple Word. La Figure 18.2 montre une portion de texte coupée dans un document Word et collée dans

La fonction suivante stocke dans le Presse-papiers une chaîne de texte identifiée comme de type RTF (Rich Text Format) :

```
private void WriteClipboard(string rtfText)
{
  DataObject data = new DataObject();
  data.SetData(DataFormats.Rtf, rtfText);
  Clipboard.SetDataObject(data, true);
}
```

La méthode `WriteClipboard()` accepte un argument `string` pour le copier dans le Presse-papiers. Elle commence par créer un objet `DataObject()`, dans lequel elle stocke la chaîne et l'indication que le texte est en fait une série de commandes RTF, et non un objet de type feuille de calcul ou base de données. La classe `DataFormats` n'est en fait rien de plus qu'un ensemble de descripteurs de différents formats de données, `DataFormats.Rtf` étant celui qui nous intéresse ici. La méthode `SetDataObject()` copie la chaîne RTF dans le Presse-papiers.

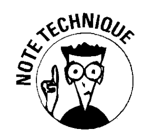
La lecture des données dans le Presse-papiers est le même processus en sens inverse, mais il vous faut y ajouter quelques tests pour garantir que la requête de lecture sera ignorée si la donnée contenue dans le Presse-papiers n'est pas de type chaîne :

```
private string ReadClipboard()
{
  // récupère le contenu du Presse-papiers
  IDataObject data = Clipboard.GetDataObject();
  if (data == null)
  {
    return null;
  }
  // une fois les données récupérées, vérifie qu'elles sont
  // au format RTF
  object o = data.GetData(DataFormats.Rtf, true);
  if (o == null)
  {
    return null;
  }
  // nous avons quelque chose, mais assurons-nous
  // que c'est bien une chaîne
  if ((o is string) == false)
  {
    return null;
  }
```

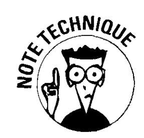
formelle, ces accessoires sont appelés des *composants*). Il suffit de choisir un composant dans la Boîte à outils, et de le faire glisser pour le déposer sur le formulaire. Vous pouvez ensuite le personnaliser en ajustant toutes les propriétés que vous voulez dans la fenêtre Propriétés, judicieusement nommée.

La fenêtre Propriétés liste deux types fondamentalement différents de propriétés. Le premier de ces ensembles, que j'appelle les *propriétés statiques*, comporte la police, la forme, la couleur d'arrière-plan, et le texte initial. Ce sont également des propriétés du point de vue du langage C#. (Pour en savoir plus sur la structure Propriété de C#, reportez-vous au Chapitre 11.)

La fenêtre Propriétés contient aussi un ensemble de propriétés complètement différent, qui correspondent plutôt aux méthodes de C#. Je les appelle *propriétés actives*.



Les propriétés actives correspondent en fait à ce que l'on appelle un *délégué*. Un délégué est une référence à un couple objet/méthode. Dans ce cas, l'objet est le composant sélectionné, et la méthode est la "propriété" de la liste des propriétés actives.



Les propriétés dynamiques sont plus communément appelées des événements. La méthode qui est invoquée lorsque l'événement se produit s'appelle un *gestionnaire d'événement*. Mais je ne veux pas introduire de complications inutiles.

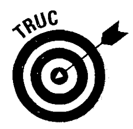
Les propriétés actives d'un objet sont les méthodes invoquées par C# lorsque certaines circonstances particulières se produisent. Par exemple, la propriété `Button.Click` est invoquée lorsque que l'utilisateur clique sur un bouton. Mais ces propriétés actives offrent un contrôle bien plus précis que cela. Par exemple, si vous voulez différencier le fait d'enfoncer un bouton ou de le relâcher, vous avez une propriété différente pour chacune de ces deux actions. Une propriété active est déclenchée quand le pointeur se trouve sur un bouton, que l'utilisateur clique ou non, et c'est une autre propriété qui passe à l'action lorsque la souris va ailleurs (c'est généralement ce qui est utilisé pour changer la couleur d'un bouton quand le pointeur passe dessus).

Pour accéder aux propriétés actives, sélectionnez le composant, et cliquez sur le bouton contenant un éclair en haut de la fenêtre Propriétés. La Figure 18.1 montre une partie des propriétés actives d'un composant `TextBox`.

Afin que le programme `SimpleEditor` fasse ce que l'on attend de lui, vous devez définir une ou plusieurs propriétés actives pour chacun de ses composants. `SimpleEditor` paraît soudain moins simple.

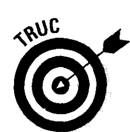
```
                                                        this.menuItem5,
                                                        this.menuItem9,
                                                      this.menuItem12});
      //
      // trackBar1
      //
      this.trackBar1.Anchor = ((System.Windows.Forms.AnchorStyles.Bottom |
System.Windows.Forms.AnchorStyles.Left)
        | System.Windows.Forms.AnchorStyles.Right);
      this.trackBar1.Location = new System.Drawing.Point(40, 248);
      this.trackBar1.Maximum = 24;
      this.trackBar1.Minimum = 8;
      this.trackBar1.Name = "trackBar1";
      this.trackBar1.Size = new System.Drawing.Size(208, 42);
      this.trackBar1.TabIndex = 2;
      this.trackBar1.Value = 12;
    }
    #endregion
  }
}
```

J'ai supprimé toutes les sections qui ne concernent pas le menu principal (`MainMenu`), l'une des options du `Menu`, et la `TrackBar`. Chacun de ces objets est un membre donnée de la classe `Form1`. Le Concepteur de formulaires crée les noms de ces membres donnée en concaténant le type de l'objet avec un numéro.



J'aurais pu vous faire définir des noms dans la fenêtre Propriétés pour obtenir quelque chose de plus parlant, mais c'était sans importance. Pour des programmes de grande taille, la définition de vos propres noms peut rendre le code qui en résulte beaucoup plus facile à lire.

La méthode `InitializeComponent()` commence par créer un objet de chaque type.



Ne vous étonnez pas du fait que le Concepteur de formulaires donne le nom complet de chaque classe, y compris son espace de nom (`System.Windows.Forms`).

Dans l'une des sections suivantes du programme, `InitializeComponent()` assigne à chacun de ces objets les propriétés que vous avez définies dans la fenêtre Propriétés.

Si vous générez à nouveau `SimpleEditor`, le redimensionnement fonctionne comme vous pouvez l'attendre, comme le montrent le "petit `SimpleEditor`" de la Figure 17.11 et le "grand `SimpleEditor`" de la Figure 17.12.

Tableau 17.2 : Type d'ancrage pour chaque composant.

Composant	Ancrage
RichTextBox	Haut, bas, gauche, droite
Zone de texte "Taille de police"	Bas
Étiquette "Taille de police"	Bas
TrackBar	Bas, gauche, droite
Étiquette de l'extrémité gauche de la TrackBar	Bas, gauche
Étiquette de l'extrémité droite de la TrackBar	Bas, droite

Figure 17.10 : Cliquez sur les bras de la fenêtre d'ancrage pour sélectionner (gris foncé) ou désélectionner (blanc) l'ancrage dans chaque direction.

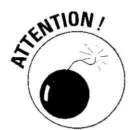
Figure 17.11 : Le petit `SimpleEditor`.

Redimensionner le formulaire

Les utilisateurs aiment que les fenêtres soient redimensionnables. Pour être plus précis, les utilisateurs peuvent avoir envie de redimensionner le formulaire de `SimpleEditor`. Par défaut, un formulaire est redimensionnable, mais les objets qu'il contient ne le sont généralement pas. C'est une chose à laquelle vous pouvez remédier, mais la solution n'est pas triviale.

Si vous voulez faire simple, ne permettez pas aux utilisateurs de redimensionner le formulaire. Ce n'est pas forcément évident, mais le redimensionnement est une fonction du cadre. L'assignation de la valeur `Fixed3D` à la propriété `FormBorderStyle` met le formulaire hors d'atteinte du zèle des souris.

1. **Sélectionnez le formulaire. Identifiez la propriété `FormBorderStyle`. Sa valeur par défaut est `Sizable`. Cliquez sur cette propriété pour faire apparaître une liste déroulante des valeurs possibles. Sélectionnez `Fixed3D` pour interdire le redimensionnement.**



La suite de cette section, consacrée à la manière de rendre `SimpleEditor` redimensionnable, peut être considérée comme appartenant au domaine technique. Vous pouvez l'ignorer et continuer à avancer. Vous y reviendrez quand vous voudrez.

Pour rendre un formulaire redimensionnable, la difficulté est de dire aux composants qu'il contient comment ils doivent répondre. Par défaut, la plupart des composants ne sont pas redimensionnables. Si vous redimensionnez le formulaire, ils resteront là où ils étaient, comme le montre la Figure 17.9.

Si `SimpleEditor` doit être redimensionnable, ses composants doivent savoir quoi faire. Par exemple, si le cadre est agrandi, la `TrackBar` doit se déplacer vers la droite, tout en suivant le bord inférieur du formulaire. Autrement dit, la `TrackBar` est ancrée en bas du formulaire. Si le formulaire est allongé verticalement, la `TrackBar` reste sur le bord inférieur.

En plus, la `TrackBar` doit s'étirer horizontalement entre le bord gauche et le bord droit du formulaire. Cet effet est produit par son ancrage sur les bords droit et gauche. Quelle que soit la largeur donnée au formulaire, la `TrackBar` ne doit jamais dépasser du formulaire, par la droite ou par la gauche.

Encore un détail : il faut centrer horizontalement la zone de texte dans le formulaire. Vous pouvez le faire visuellement, mais il y a une meilleure solution.

5. **Sélectionnez la zone de texte, et cliquez sur l'outil Centrer horizontalement dans la barre d'outils Disposition.**

   La zone de texte est centrée horizontalement dans le formulaire.

6. **Venons-en maintenant à notre `TrackBar`. Sélectionnez le composant `TrackBar` dans la Boîte à outils, et placez-le tout en bas du formulaire `SimpleEditor`.**

   La taille verticale d'un composant `TrackBar` est fixe, mais vous pouvez l'étirer horizontalement pour lui donner la longueur que vous trouverez raisonnable. Encore une fois, "raisonnable" est une question de préférence personnelle, et vous pourrez modifier cette longueur lorsque vous aurez vu ce qu'elle donne en pratique.

   Une `TrackBar` possède plusieurs propriétés de comportement intéressantes. En fonctionnement, `SimpleEditor` aura besoin de demander à la `TrackBar` la valeur qu'elle contient. Ce qui soulève la question : "À quelle valeur correspondent la position la plus à droite et la position la plus à gauche de l'index ?" Ces deux valeurs sont définies respectivement par les propriétés `Minimum` et `Maximum` de `TrackBar`. Nous avons dit que la taille de la police peut aller de 8 à 24 points.

7. **Sélectionnez la `TrackBar` (si elle ne l'est pas déjà). Assignez la valeur 8 à la propriété `Minimum`, et la valeur 24 à la propriété `Maximum`. Assignez la valeur 12 à la propriété `Value`.**

8. **Centrez la `TrackBar` : sélectionnez-la, et cliquez sur le bouton Centrer horizontalement dans la barre d'outils Disposition.**

Encore un coup de peinture et nous y sommes

Les composants dont nous avions besoin pour la taille de la police sont en place et prêts à fonctionner, mais ils pourraient être plus sympathiques. Vous et moi, nous savons à quoi ils servent, mais personne d'autre ne va le deviner. `SimpleEditor` a besoin de quelques étiquettes pour indiquer à quoi servent les différents champs.

Tableau 17.1 : Raccourcis clavier des éléments de menu.

Élément de menu	Raccourci
Fichier/Ouvrir	CtrlO
Fichier/Enregistrer	CtrlS
Fichier/Quitter	CtrlQ
Édition/Couper	CtrlX
Édition/Copier	CtrlC
Édition/Coller	CtrlV
Format/Gras	CtrlB
Format/Italique	CtrlI
?	F1

Ajouter les contrôles d'ajustement de la police

`SimpleEditor` doit aussi être capable de modifier la police, dans certaines limites arbitraires. Dans cette section, vous allez ajouter une zone de texte dans laquelle l'utilisateur pourra taper la taille qu'il veut donner à la police. `SimpleEditor` disposera aussi d'un contrôle analogue, que l'on appelle `TrackBar`, dans lequel l'utilisateur peut faire glisser un index d'une extrémité à l'autre pour augmenter ou diminuer la taille de la police.



En dehors de rendre `SimpleEditor` plus facile à utiliser, cette fonctionnalité permet aussi de montrer comment relier deux contrôles.

1. **Ouvrez la Boîte à outils, et faites glisser un contrôle `TextBox` en bas de la fenêtre `SimpleEditor`. Comme la taille par défaut est un peu grande pour deux chiffres, vous pouvez la réduire horizontalement.**

   Il y a beaucoup de choses dans la propriété `Font` (police) : la police elle-même et sa taille, ainsi que des propriétés comme `Bold` (gras), `Italic` (italique), et `Strikeout` (barré). C'est pour cette raison qu'il y a un petit signe plus à gauche de la propriété `Font`. Cliquez sur ce signe plus, et vous voyez apparaître toutes les propriétés que contient `Font`, comme le montre la Figure 17.7. (Et le signe plus devient un signe moins – si vous cliquez sur le signe moins, vous ne voyez plus que la propriété `Font`, comme avant.)

2. **Suivez les instructions simples qui apparaissent dans le Concepteur : cliquez sur les mots** `Tapez ici`**, puis tapez le nom de votre premier élément de menu :** Fichier.

   Le Concepteur répond en affichant un autre cadre `Tapez ici` au-dessous, et encore un autre à droite du premier, comme le montre la Figure 17.5. C'est tellement excitant que je ne sais pas par lequel commencer.

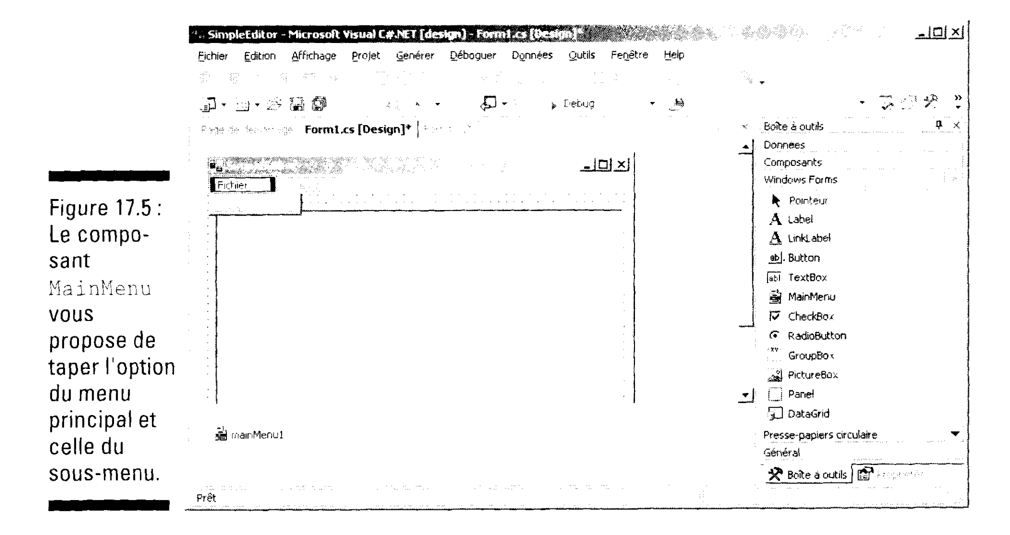


Figure 17.5 : Le composant `MainMenu` vous propose de taper l'option du menu principal et celle du sous-menu.

3. **Cliquez dans le cadre** `Tapez ici` **au-dessous de Fichier, et entrez les trois options du menu Fichier :** Ouvrir, Enregistrer, **et** Quitter.

4. **Cela fait, cliquez dans le cadre** `Tapez ici` **à droite du menu Fichier, et entrez Édition et ses options :** Couper, Copier, **et** Coller.

5. **Déplacez-vous à nouveau d'un cran vers la droite, et ajoutez** Format **et ses options :** Gras **et** Italique.

6. **Enfin, ajoutez ?** (l'aide) **à la barre de menus.**

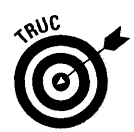
   Remarquez la nouvelle zone ouverte par le Concepteur au-dessous de la zone de dessins avec la création de votre premier menu principal. Vous pouvez cliquer sur l'objet `mainMenu1` qui y apparaît pour définir les propriétés d'ensemble de ces menus. Vous pouvez aussi utiliser cette zone pour y placer les objets qui ne sont pas directement visibles (par exemple, une boîte de dialogue qui apparaît seulement dans certaines circonstances).

vous-même. Tous ces sujets sont fort intéressants, mais, comme vous vous en doutez, sortent du cadre de ce livre.

En faisant défiler vers le haut et vers le bas le contenu de la Boîte à outils, vous découvrez une pléthore de composants. On y trouve des étiquettes, des boutons, des zones de texte, des menus, et une quantité d'autres objets graphiques. Il y en a sûrement un qui est ce dont nous avons besoin pour la fenêtre d'édition. On pourrait penser que le composant `TextBox` est ce qu'il nous faut, mais une zone de texte est plutôt adaptée à la saisie d'un texte simple (en général, une ligne). Par exemple, vous utiliserez une zone de texte pour permettre à l'utilisateur d'entrer un simple nombre entier pour la taille de police.

En fait, le meilleur choix pour la fenêtre d'édition est le composant `RichTextBox`. Celui-ci permet de saisir et d'afficher du texte dans le format nommé RTF (Rich Text Format). Un fichier RTF est semblable à un fichier au format Microsoft Word (.DOC), à cette différence que RTF est plutôt un standard. Ce format a toutes les propriétés dont nous avons besoin : italique, gras, différentes tailles de police, et il est pris en compte par la plupart des traitements de texte pour Windows, dont Word, et d'autres traitements de texte écrits pour d'autres systèmes d'exploitation, par exemple pour Unix.

2. **Afin de créer la fenêtre d'édition, cliquez sur le symbole `RichTextBox` dans la Boîte à outils. Placez le pointeur dans le coin supérieur gauche du formulaire Simple Editor, puis maintenez enfoncé le bouton gauche de la souris pour le faire glisser vers le bas et vers la droite, créant ainsi une zone d'édition comme celle que montre la Figure 17.4.**



Ne vous préoccupez pas trop de la taille et de l'emplacement exact de la zone `RichTextBox`. Vous pourrez toujours la déplacer et la redimensionner autant que nécessaire.

3. **Je n'aime pas beaucoup le texte initial de `richTextBox1`. Pour le modifier, ouvrez la fenêtre Propriétés, et remplacez le contenu de la propriété `Text` par rien. Autrement dit, effacez ce qui s'y trouve pour laisser ce champ vierge.**

Le texte disparaît de la zone `RichTextBox`.



La même propriété peut être interprétée de façon différente par deux composants différents. La propriété `Text` en est le meilleur exemple. Pour un formulaire, c'est l'étiquette qui se trouve dans la barre de titre. Pour une zone de texte (`TextBox` ou `RichTextBox`),

Un livre simple, clair et drôle pour découvrir C# !

C# POUR LES NULS

La première collection informatique dans le monde

A mettre entre toutes les mains !



au quotidien !

Pour comprendre enfin quelque chose à la micro-informatique !

Vous voici confronté à un micro-ordinateur – plus par nécessité que par goût, avouons-le –, sans savoir par quel bout prendre cet instrument barbare et capricieux. Oubliez toute appréhension, cette nouvelle collection est réellement faite pour vous !

Le nouveau C de Microsoft, pierre angulaire de la solution .NET

Grâce à ce livre, vous allez rapidement écrire vos premières applications en C#, sans pour autant devenir un gourou de la programmation. C#, c'est le nouveau langage de programmation développé par Microsoft, et qui se présente comme la pierre angulaire de la solution .NET du géant du logiciel. Rassurez-vous, on ne vous assommera pas avec toutes les subtilités du langage, mais vous posséderez les bases nécessaires pour utiliser la panoplie d'outils du parfait programmeur C#.

L'ESPRIT DES NULS

L'informatique en français dans le texte.

Tout et seulement tout ce que vous devez savoir.

Un accès rapide à l'information grâce à un système d'icônes d'aide à la navigation.

Les dix commandements.

Une bonne dose d'humour.

DÉCOUVREZ

Variables et opérateurs.

Programmation et objets.

La programmation orientée objet.

Classe, héritage et polymorphisme.

Programmer sous Windows avec Visual Studio.

SUIVEZ LE GUIDE



Au vu de ce symbole, si vous êtes allergique à la technique, passez votre chemin.



Cette icône signale une manipulation qui va vous simplifier la vie.



Désolé, il faut quand même retenir ceci.

65 3303 8
ISBN-2-84427-259-2



21,90 € (143,65 F)

Hungry Minds™





POUR LES NULS

65 3303 8

